MARK BATES • CORY LANOU • TIMOTHY J. RAYMOND



HOW TO CODE IN GO



ISBN 978-0-9997730-6-2

How To Code in Go

Mark Bates, Cory LaNou, Tim Raymond

DigitalOcean, New York City, New York, USA

2020-06

How To Code in Go

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Preface — Getting Started with this Book

We recommend that you begin with a clean, new server to start learning how to program with Go. You can also use a local computer like a laptop or desktop as well. One of the benefits of Go is that most of the code that you write will compile and run on Linux, Windows, and macOS, so you can choose a development environment that suits you.

If you are unfamiliar with Go, or do not have a development environment set up, Chapter 1 of this book goes into detail about how to install Go for development on a local Ubuntu 18.04 system, Chapter 2 explains how to set up Go on macOS, and Chapter 3 covers how to get started with Go using Windows 10.

Once you are set up with a local or remote development environment, you will be able to follow along with each chapter at your own pace, and in the order that you choose.

Introduction

About this Book

Go (or GoLang) is a modern programming language originally developed by Google that uses high-level syntax similar to scripting languages. It is popular for its minimal syntax and innovative handling of concurrency, as well as for the tools it provides for building native binaries on foreign platforms.

All of the chapters in this book are written by members of the Gopher Guides team. Individually, each team member is engaged with their local software development communities with hosting and organizing meetups and conferences. Collectively, the Gopher Guides Team is committed to educating Go developers, and fostering an inclusive and supportive global community of developers that welcomes beginners and experts alike.

Motivation for this Book

Many books and other educational resources about Go rely on building progressively more complex examples to illustrate concepts like composition, testing, and packaging. This book explores various Go topics with clear, self-contained example code.

Structuring the book this way means that the examples in each chapter are concise and specific to the topic of the chapter. For those who are starting out learning about Go, the chapters are ordered in a way that should guide you from a beginning “Hello, World” program all the way to parsing command line flags with your programs.

For more experienced programmers who would like to use this book as a reference, or readers who like to choose what to focus on, this approach of loosely related chapters also means that you can skip between chapters to learn about particular concepts that interest you without missing out on context from preceding chapters.

Learning Goals and Outcomes

This book starts with chapters on how to set up and configure a local Go development environment. It ends with chapters that explain how to build your Go programs with conditional imports to target specific operating systems, and a final chapter on parsing command line flags to help you build your own utilities.

By the end of this book you will be able to write programs that use conditional logic with switch statements, define your own data structures, implement interfaces to reuse portions of your code, and write custom error handling functions.

These examples are just a small sample of the topics that are covered in the book. We hope that you continue learning about Go once you are finished reading it. Go is a fun, powerful language that is gaining in popularity. We also hope that this book enables you to become a more productive Go programmer, and that it helps you learn how to build useful software with Go for yourself or others.

How To Install Go and Set Up a Local Programming Environment on Ubuntu 18.04

Written by Gopher Guides

Go is a programming language that was born out of frustration at Google. Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production. Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.

While Go is a versatile programming language that can be used for many different programming projects, it's particularly well suited for networking/distributed systems programs, and has earned a reputation as "the language of the cloud". It focuses on helping the modern programmer do more with a strong set of tooling, removing debates over formatting by making the format part of the language specification, as well as making deployment easy by compiling to a single binary. Go is easy to learn, with a very small set of keywords, which makes it a great choice for beginners and experienced developers alike.

This tutorial will guide you through installing and configuring a programming workspace with Go via the

command line. This tutorial will explicitly cover the installation procedure for Ubuntu 18.04, but the general principles can apply to other Debian Linux distributions.

Prerequisites

You will need a computer or virtual machine with Ubuntu 18.04 installed, as well as have administrative access to that machine and an internet connection. You can download this operating system via the Ubuntu 18.04 releases page.

Step 1 — Setting Up Go

In this step, you'll install Go by downloading the current release from the official Go downloads page.

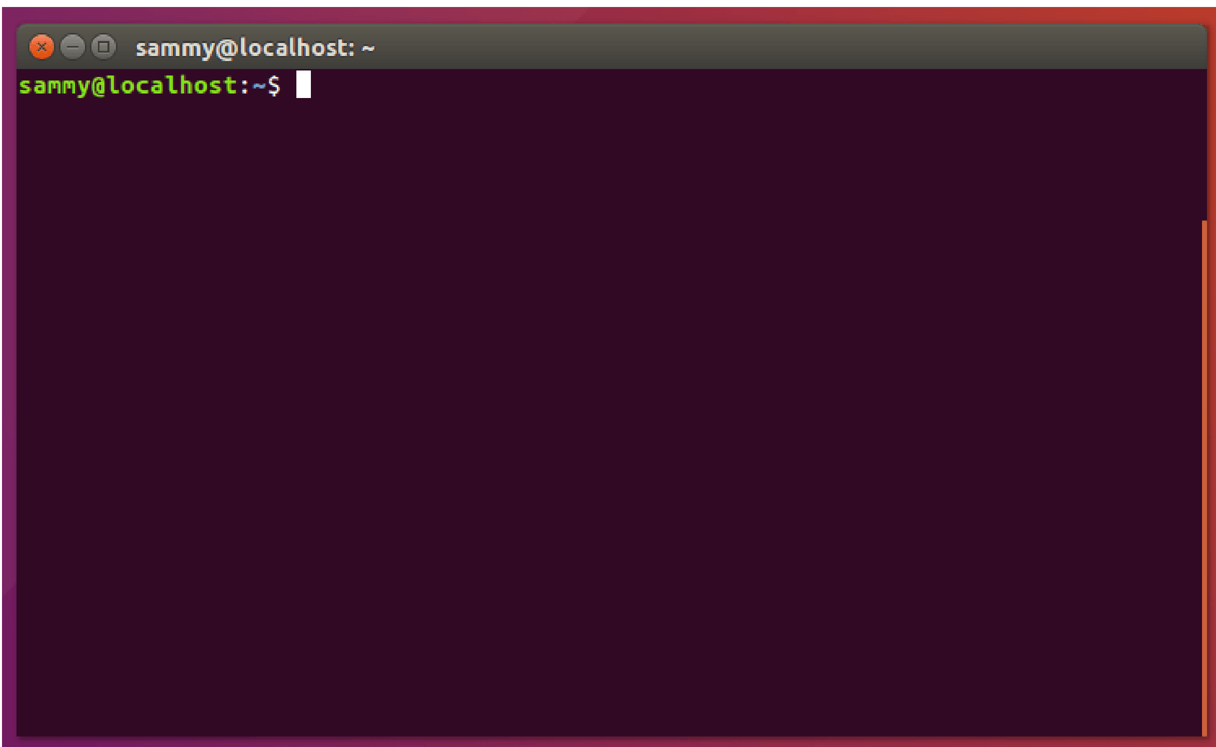
To do this, you'll want to find the URL for the current binary release tarball. You will also want to note the SHA256 hash listed next to it, as you'll use this hash to verify the downloaded file.

You'll be completing the installation and setup on the command line, which is a non-graphical way to interact with your computer. That is, instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well.

The command line, also known as a shell or terminal, can help you modify and automate many of the tasks you do on a computer every day, and is an essential tool for software developers. There are many terminal commands to learn

that can enable you to do more powerful things. For more information about the command line, check out the [Introduction to the Linux Terminal] tutorial.

On Ubuntu 18.04, you can find the Terminal application by clicking on the Ubuntu icon in the upper-left hand corner of your screen and typing `terminal` into the search bar. Click on the Terminal application icon to open it. Alternatively, you can hit the `CTRL`, `ALT`, and `T` keys on your keyboard at the same time to open the Terminal application automatically.



Ubuntu Terminal

Once the terminal is open, you will manually install the Go binaries. While you could use a package manager, such

as `apt-get`, walking through the manual installation steps will help you understand any configuration changes to your system that are needed to have a valid Go workspace.

Before downloading Go, make sure that you are in the home (~) directory:

```
cd ~
```

Use `curl` to retrieve the tarball URL that you copied from the official Go downloads page:

```
curl -O https://dl.google.com/go/go1.12.1.linux-amd64.tar.gz
```

Next, use `sha256sum` to verify the tarball:

```
sha256sum go1.12.1.linux-amd64.tar.gz
```

The hash that is displayed from running the above command should match the hash that was on the downloads page. If it does not, then this is not a valid file and you should download the file again.

Output

```
2a3fdabf665496a0db5f41ec6af7a9b15a49fbe71a85a50ca3
8b1f13a103aeec  go1.12.1.linux-amd64.tar.gz
```

Next, extract the downloaded archive and install it to the desired location on the system. It's considered best practice to keep it under `/usr/local`:

```
sudo tar -xvf go1.12.1.linux-amd64.tar.gz -C
/usr/local
```

You will now have a directory called `go` in the `/usr/local` directory. Next, recursively change this directory's owner and group to root:

```
sudo chown -R root:root /usr/local/go
```

This will secure all the files and ensure that only the root user can run the Go binaries.

Note: Although `/usr/local/go` is the officially-recommended location, some users may prefer or require different paths.

In this step, you downloaded and installed Go on your Ubuntu 18.04 machine. In the next step you will configure your Go workspace.

Step 2 — Creating Your Go Workspace

You can create your programming workspace now that Go is installed. The Go workspace will contain two directories at its root:

- `src`: The directory that contains Go source files. A source file is a file that you write using the Go programming language. Source files are used by the Go compiler to create an executable binary file.
- `bin`: The directory that contains executables built and installed by the Go tools. Executables are binary files that run on your system and execute tasks. These are typically the programs compiled by your source code or other downloaded Go source code.

The `src` subdirectory may contain multiple version control repositories (such as [Git](), [Mercurial](), and [Bazaar]()). This allows for a canonical import of code in your project. Canonical imports are imports that reference a fully qualified package, such as `github.com/digitalocean/godo`.

You will see directories like `github.com`, `golang.org`, or others when your program imports third party libraries. If you are using a code repository like `github.com`, you will also put your projects and source files under that directory. We will explore this concept later in this step.

Here is what a typical workspace may look like:

```
.
├── bin
│   ├── buffalo
# command executable
│   ├── dlv
# command executable
│   └── packr
# command executable
└── src
    └── github.com
        └── digitalocean
            └── godo
                ├── .git
# Git repository metadata
                ├── account.go
```

```
# package source
                ├── account_test.go
# test source
                ├── ...
                ├── timestamp.go
                ├── timestamp_test.go
                └── util
                    ├── droplet.go
                    └── droplet_test.go
```

The default directory for the Go workspace as of 1.8 is your user's home directory with a `go` subdirectory, or `$HOME/go`. If you are using an earlier version of Go than 1.8, it is still considered best practice to use the `$HOME/go` location for your workspace.

Issue the following command to create the directory structure for your Go workspace:

```
mkdir -p $HOME/go/{bin,src}
```

The `-p` option tells `mkdir` to create all `parents` in the directory, even if they don't currently exist. Using `{bin,src}` creates a set of arguments to `mkdir` and tells it to create both the `bin` directory and the `src` directory.

This will ensure the following directory structure is now in place:

```
└── $HOME
    └── go
        ├── bin
        └── src
```

Prior to Go 1.8, it was required to set a local environment variable called `$GOPATH`. `$GOPATH` told the compiler where to find imported third party source code, as well as any local source code you had written. While it is no longer explicitly required, it is still considered a good practice as many third party tools still depend on this variable being set.

You can set your `$GOPATH` by adding the global variables to your `~/.profile`. You may want to add this into `.zshrc` or `.bashrc` file as per your shell configuration.

First, open `~/.profile` with `nano` or your preferred text editor:

```
nano ~/.profile
```

Set your `$GOPATH` by adding the following to the file:

~/.profile

```
export GOPATH=$HOME/go
```

When Go compiles and installs tools, it will put them in the `$GOPATH/bin` directory. For convenience, it's common to add the workspace's `/bin` subdirectory to your `PATH` in your `~/.profile`:

~/.profile

```
export PATH=$PATH:$GOPATH/bin
```

This will allow you to run any programs you compile or download via the Go tools anywhere on your system.

Finally, you need to add the `go` binary to your `PATH`. You can do this by adding `/usr/local/go/bin` to the end of the line:

~/.profile

```
export PATH=$PATH:$GOPATH/bin:/usr/local/go/bin
```

Adding `/usr/local/go/bin` to your `$PATH` makes all of the Go tools available anywhere on your system.

To update your shell, issue the following command to load the global variables:

```
. ~/.profile
```

You can verify your `$PATH` is updated by using the `echo` command and inspecting the output:

```
echo $PATH
```

You will see your `$GOPATH/bin` which will show up in your home directory. If you are logged in as `root`, you would see `/root/go/bin` in the path.

Output

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/
root/go/bin:/usr/local/go/bin
```

You will also see the path to the Go tools for `/usr/local/go/bin`:

Output

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/
root/go/bin:/usr/local/go/bin
```

Verify the installation by checking the current version of Go:

```
go version
```

And we should receive output like this:

Output

```
go version go1.12.1 linux/amd64
```

Now that you have the root of the workspace created and your `$GOPATH` environment variable set, you can create your future projects with the following directory structure. This example assumes you are using `github.com` as your repository:

```
$GOPATH/src/github.com/username/project
```

So as an example, if you were working on the https://github.com/digitalocean/godo project, it would be stored in the following directory:

```
$GOPATH/src/github.com/digitalocean/godo
```

This project structure will make projects available with the `go get` tool. It will also help readability later. You can verify this by using the `go get` command and fetch the `godo` library:

```
go get github.com/digitalocean/godo
```

This will download the contents of the `godo` library and create the `$GOPATH/src/github.com/digitalocean/godo` directory on your machine.

You can check to see if it successfully downloaded the `godo` package by listing the directory:

```
ll $GOPATH/src/github.com/digitalocean/godo
```

You should see output similar to this:

Output

```
drwxr-xr-x 4 root root  4096 Apr  5 00:43 ./
drwxr-xr-x 3 root root  4096 Apr  5 00:43 ../
drwxr-xr-x 8 root root  4096 Apr  5 00:43 .git/
-rwxr-xr-x 1 root root     8 Apr  5 00:43
.gitignore*
-rw-r--r-- 1 root root    61 Apr  5 00:43
.travis.yml
-rw-r--r-- 1 root root  2808 Apr  5 00:43
CHANGELOG.md
-rw-r--r-- 1 root root  1851 Apr  5 00:43
CONTRIBUTING.md
.
.
.
-rw-r--r-- 1 root root  4893 Apr  5 00:43 vpcs.go
-rw-r--r-- 1 root root  4091 Apr  5 00:43
vpcs_test.go
```

In this step, you created a Go workspace and configured the necessary environment variables. In the next step you will test the workspace with some code.

Step 3 — Creating a Simple Program

Now that you have the Go workspace set up, create a “Hello, World!” program. This will make sure that the

workspace is configured properly, and also gives you the opportunity to become more familiar with Go. Because we are creating a single Go source file, and not an actual project, we don't need to be in our workspace to do this.

From your home directory, open up a command-line text editor, such as `nano`, and create a new file:

```
nano hello.go
```

Write your program in the new file:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

This code will use the `fmt` package and call the `Println` function with `Hello, World!` as the argument. This will cause the phrase `Hello, World!` to print out to the terminal when the program is run.

Exit `nano` by pressing the `CTRL` and `X` keys. When prompted to save the file, press `Y` and then `ENTER`.

Once you exit out of `nano` and return to your shell, run the program:

```
go run hello.go
```

The `hello.go` program will cause the terminal to produce the following output:

Output

```
Hello, World!
```

In this step, you used a basic program to verify that your Go workspace is properly configured.

Conclusion

Congratulations! At this point you have a Go programming workspace set up on your Ubuntu machine and can begin a coding project!

How To Install Go and Set Up a Local Programming Environment on macOS

Written by Gopher Guides

Go is a programming language that was born out of frustration at Google. Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production. Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.

While Go is a versatile programming language that can be used for many different programming projects, it's particularly well suited for networking/distributed systems programs, and has earned a reputation as "the language of the cloud." It focuses on helping the modern programmer do more with a strong set of tooling, removing debates over formatting by making the format part of the language specification, as well as making deployment easy by compiling to a single binary. Go is easy to learn, with a very small set of keywords, which makes it a great choice for beginners and experienced developers alike.

This tutorial will guide you through installing Go on your local macOS machine and setting up a programming workspace via the command line.

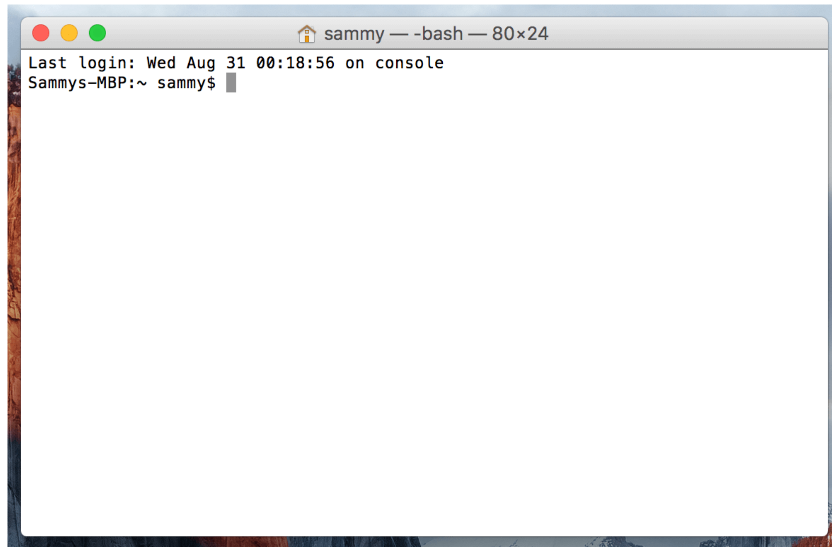
Prerequisites

You will need a macOS computer with administrative access that is connected to the internet.

Step 1 — Opening Terminal

We'll be completing most of our installation and setup on the command line, which is a non-graphical way to interact with your computer. That is, instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well. The command line, also known as a shell, can help you modify and automate many of the tasks you do on a computer every day, and is an essential tool for software developers.

The macOS Terminal is an application you can use to access the command line interface. Like any other application, you can find it by going into Finder, navigating to the Applications folder, and then into the Utilities folder. From here, double-click the Terminal like any other application to open it up. Alternatively, you can use Spotlight by holding down the `CMD` and `SPACE` keys to find Terminal by typing it out in the box that appears.



macOS Terminal

There are many more Terminal commands to learn that can enable you to do more powerful things. The article "[An Introduction to the Linux Terminal] (https://www.digitalocean.com/community/tutorials/an-introduction-to-the-linux-terminal)" can get you better oriented with the Linux Terminal, which is similar to the macOS Terminal.

Now that you have opened up Terminal, you can download and install Xcode, a package of developer tools that you will need in order to install Go.

Step 2 — Installing Xcode

Xcode is an integrated development environment (IDE) that is comprised of software development tools for macOS. You can check if Xcode is already installed by typing the following in the Terminal window:

```
xcode-select -p
```

The following output means that Xcode is installed:

Output

```
/Library/Developer/CommandLineTools
```

If you received an error, then in your web browser install [Xcode from the App Store] (https://itunes.apple.com/us/app/xcode/id497799835?mt=12&ign-mpt=uo%3D2) and accept the default options.

Once Xcode is installed, return to your Terminal window. Next, you'll need to install Xcode's separate Command Line Tools app, which you can do by typing:

```
xcode-select --install
```

At this point, Xcode and its Command Line Tools app are fully installed, and we are ready to install the package manager Homebrew.

Step 3 — Installing and Setting Up Homebrew

While the macOS Terminal has a lot of the functionality of Linux Terminals and other Unix systems, it does not ship with a package manager that accommodates best practices. A package manager is a collection of software tools that work to automate installation processes that include initial software installation, upgrading and configuring of software, and removing software as needed. They keep installations in a central location and can maintain all software packages on the system in formats that are commonly used. Homebrew provides macOS with a free and open source software package managing system that simplifies the installation of software on macOS.

To install Homebrew, type this into your Terminal window:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Homebrew is made with Ruby, so it will be modifying your computer's Ruby path. The `curl` command pulls a script from the specified URL. This script will explain what it will do and then pauses the process to prompt you to confirm. This provides you with a lot of feedback on what the script is going to be doing to your system and gives you the opportunity to verify the process.

If you need to enter your password, note that your keystrokes will not display in the Terminal window but they will be recorded. Simply press the `return` key once you've entered your password. Otherwise press the letter `y` for "yes" whenever you are prompted to confirm the installation.

Let's walk through the flags that are associated with the `curl` command:

- The `-f` or `--fail` flag tells the Terminal window to give no HTML document output on server errors.
- The `-s` or `--silent` flag mutes `curl` so that it does not show the progress meter, and combined with the `-S` or `--show-error` flag it will ensure that `curl` shows an error message if it fails.
- The `-L` or `--location` flag will tell `curl` to redo the request to a new place if the server reports that the requested page has moved to a different location.

Once the installation process is complete, we'll put the Homebrew directory at the top of the `PATH` environment variable. This will ensure that Homebrew installations will be called over the tools that macOS may select automatically that could run counter to the development environment we're creating.

You should create or open the `~/.bash_profile` file with the command-line text editor nano using the `nano` command:

```
nano ~/.bash_profile
```

Once the file opens up in the Terminal window, write the following:

```
export PATH=/usr/local/bin:$PATH
```

To save your changes, hold down the `CTRL` key and the letter `o`, and when prompted press the `RETURN` key. Now you can exit nano by holding the `CTRL` key and the letter `x`.

Activate these changes by executing the following in Terminal:

```
source ~/.bash_profile
```

Once you have done this, the changes you have made to the `PATH` environment variable will go into effect.

You can make sure that Homebrew was successfully installed by typing:

```
brew doctor
```

If no updates are required at this time, the Terminal output will read:

Output

```
Your system is ready to brew.
```

Otherwise, you may get a warning to run another command such as `brew update` to ensure that your installation of Homebrew is up to date.

Once Homebrew is ready, you can install Go.

##Step 4 — Installing Go

You can use Homebrew to search for all available packages with the `brew search` command. For the purpose of this tutorial, you will search for Go-related packages or modules:

```
brew search golang
```

Note: This tutorial does not use `brew search go` as it returns too many results. Because `go` is such a small word and would match many packages, it has become common to use `golang` as the search term. This is common practice when searching the internet for Go-related articles as well. The term Golang was born from the domain for Go, which is `golang.org`.

The Terminal will output a list of what you can install:

Output

```
golang  golang-migrate
```

Go will be among the items on the list. Go ahead and install it:

```
brew install golang
```

The Terminal window will give you feedback regarding the installation process of Go. It may take a few minutes before installation is complete.

To check the version of Go that you installed, type the following:

```
go version
```

This will output the specific version of Go that is currently installed, which will by default be the most up-to-date, stable version of Go that is available.

In the future, to update Go, you can run the following commands to first update Homebrew and then update Go. You don't have to do this now, as you just installed the latest version:

```
brew update
brew upgrade golang
```

`brew update` will update the formulae for Homebrew itself, ensuring you have the latest information for packages you want to install. `brew upgrade golang` will update the `golang` package to the latest release of the package.

It is good practice to ensure that your version of Go is up-to-date.

With Go installed on your computer, you are now ready to create a workspace for your Go projects.

Step 5 — Creating Your Go Workspace

Now that you have Xcode, Homebrew, and Go installed, you can go on to create your programming workspace.

The Go workspace will contain two directories at its root:

- `src`: The directory that contains Go source files. A source file is a file that you write using the Go programming language. Source files are used by the Go compiler to create an executable binary file.
- `bin`: The directory that contains executables built and installed by the Go tools. Executables are binary files that run on your system and execute tasks. These are typically the programs compiled by your source code or another downloaded Go source code.

The `src` subdirectory may contain multiple version control repositories (such as Git, Mercurial, and Bazaar). You will see directories like `github.com` or `golang.org` when your program imports third party libraries. If you are using a code repository like `github.com`, you will also put your projects and source files under that directory. This allows for a canonical import of code in your project. Canonical imports are imports that reference a fully qualified package, such as `github.com/digitalocean/godo`.

Here is what a typical workspace may look like:

```
.
├── bin
│   ├── buffalo                                      # command executable
│   ├── dlv                                          # command executable
│   └── packr                                        # command executable
└── src
    └── github.com
```

```
        └── digitalocean
            └── godo
                ├── .git                            # Git reposistory
metadata
                ├── account.go                      # package source
                ├── account_test.go                 # test source
                ├── ...
                ├── timestamp.go
                ├── timestamp_test.go
                └── util
                    ├── droplet.go
                    └── droplet_test.go
```

The default directory for the Go workspace as of 1.8 is your user's home directory with a `go` subdirectory, or `$HOME/go`. If you are using a version of Go earlier than 1.8, it is considered best practice to still use the `$HOME/go` location for your workspace.

Issue the following command to create the directory structure for your Go workspace:

```
mkdir -p $HOME/go/{bin,src}
```

The `-p` option tells `mkdir` to create all `parents` in the directory, even if they don't currently exist. Using `{bin,src}` creates a set of arguments to `mkdir` and tells it to create both the `bin` directory and the `src` directory.

This will ensure the following directory structure is now in place:

```
└── $HOME
    └── go
        ├── bin
        └── src
```

Prior to Go 1.8, it was required to set a local environment variable called `$GOPATH`. While it is no longer explicitly required to do so, it is still considered a good practice as many third party tools still depend on this variable being set.

You can set your `$GOPATH` by adding it to your `~/.bash_profile`.

First, open `~/.bash_profile` with `nano` or your preferred text editor:

```
nano ~/.bash_profile
```

Set your `$GOPATH` by adding the following to the file:

~/.bash_profile

```
export GOPATH=$HOME/go
```

When Go compiles and installs tools, it will put them in the `$GOPATH/bin` directory. For convenience, it's common to add the workspace's `/bin` subdirectory to your `PATH` in your `~/.bash_profile`:

~/.bash_profile

```
export PATH=$PATH:$GOPATH/bin
```

You should now have the following entries in your `~/.bash_profile`:

~/.bash_profile

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

This will now allow you to run any programs you compile or download via the Go tools anywhere on your system.

To update your shell, issue the following command to load the global variables you just created:

```
. ~/.bash_profile
```

You can verify your `$PATH` is updated by using the `echo` command and inspecting the output:

```
echo $PATH
```

You should see your `$GOPATH/bin` which will show up in your home directory. If you were logged in as `sammy`, you would see `/Users/sammy/go/bin` in the path.

Output

```
<^>/Users/sammy/go/bin<^>:/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sl
```

Now that you have the root of the workspace created and your `$GOPATH` environment variable set, you will create your future projects with the following directory structure. This example assumes you are using github.com as your repository:

```
$GOPATH/src/github.com/username/project
```

If you were working on the https://github.com/digitalocean/godo project, you would put it in the following directory:

```
$GOPATH/src/github.com/digitalocean/godo
```

Structuring your projects in this manner will make projects available with the `go get` tool. It will also help readability later.

You can verify this by using the `go get` command to fetch the `godo` library:

```
go get github.com/digitalocean/godo
```

We can see it successfully downloaded the `godo` package by listing the directory:

```
ls -l $GOPATH/src/github.com/digitalocean/godo
```

You will receive output similar to this:

Output

```
-rw-r--r--  1 sammy  staff   2892 Apr  5 15:56 CHANGELOG.md
-rw-r--r--  1 sammy  staff   1851 Apr  5 15:56 CONTRIBUTING.md
.
.
.
-rw-r--r--  1 sammy  staff   4893 Apr  5 15:56 vpcs.go
-rw-r--r--  1 sammy  staff   4091 Apr  5 15:56 vpcs_test.go
```

In this step, you created a Go workspace and configured the necessary environment variables. In the next step you will test the workspace with some code.

Step 6 — Creating a Simple Program

Now that you have your Go workspace set up, it's time to create a simple "Hello, World!" program. This will make sure that your workspace is working and gives you the opportunity to become more familiar with Go.

Because you are creating a single Go source file, and not an actual project, you don't need to be in your workspace to do this.

From your home directory, open up a command-line text editor, such as `nano`, and create a new file:

```
nano hello.go
```

Once the text file opens up in Terminal, type out your program:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y`.

This code will use the `fmt` package and call the `Println` function with `Hello, World!` as the argument. This will cause the phrase `Hello, World!` to print out to the terminal when the program is run.

Once you exit out of `nano` and return to your shell, run the program:

```
go run hello.go
```

The `hello.go` program that you just created will cause Terminal to produce the following output:

Output

```
Hello, World!
```

In this step, you used a basic program to verify that your Go workspace is properly configured.

Conclusion

Congratulations! At this point you have a Go programming workspace set up on your local macOS machine and can begin a coding project!

How To Install Go and Set Up a Local Programming Environment on Windows 10

Written by Gopher Guides

Go is a programming language that was born out of frustration at Google. Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production. Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.

While Go is a versatile programming language that can be used for many different programming projects, it’s particularly well suited for networking/distributed systems programs, and has earned a reputation as “the language of the cloud”. It focuses on helping the modern programmer do more with a strong set of tooling, removing debates over formatting by making the format part of the language specification, as well as making deployment easy by compiling to a single binary. Go is easy to learn, with a very small set of keywords, which makes it a great choice for beginners and experienced developers alike.

This tutorial will guide you through installing Go on your local Windows 10 machine and setting up a programming

environment via the command line.

Prerequisites

You will need a Windows 10 machine with administrative access that is connected to the internet.

Step 1 — Opening and Configuring PowerShell

You'll be completing most of the installation and setup on a command-line interface, which is a non-graphical way to interact with your computer. That is, instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well. The command line, also known as a shell, can help you modify and automate many of the tasks you do on a computer every day, and is an essential tool for software developers.

PowerShell is a program from Microsoft that provides a command-line shell interface. Administrative tasks are performed by running cmdlets, pronounced command-lets, which are specialized classes of the .NET software framework that can carry out operations. Open-sourced in August 2016, PowerShell is now available across platforms, for both Windows and UNIX systems (including Mac and Linux).

To find Windows PowerShell, you can right-click on the Start menu icon on the lower left-hand corner of your screen. When the menu pops up, click on Search, and then

type `PowerShell` into the search bar. When you are presented with options, right-click on Windows PowerShell from the Desktop app. For the purposes of this tutorial, select Run as Administrator. When you are prompted with a dialog box that asks Do you want to allow this app to make changes to your PC? click on Yes.

Once you do this, you'll see a text-based interface that has a string of words that looks like this:

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> _
```

Windows 10 PowerShell

Switch out of the system folder by typing the following command:

```
cd ~
```

You'll then be in a home directory such as `PS C:\Users\sammy`.

To continue with the installation process, you must first set up permissions through PowerShell. Configured to run in the most secure mode by default, there are a few levels of permissions that you can set up as an administrator:

- Restricted is the default execution policy. Under this mode you will not be able to run scripts, and PowerShell will work only as an interactive shell.
- AllSigned will enable you to run all scripts and configuration files that are signed by a trusted publisher, meaning that you could potentially open your machine up to the risk of running malicious scripts that happen to be signed by a trusted publisher.
- RemoteSigned will let you run scripts and configuration files downloaded from the internet signed by trusted publishers, again opening your machine up to vulnerabilities if these trusted scripts are actually malicious.
- Unrestricted will run all scripts and configuration files downloaded from the internet as soon as you confirm that you understand that the file was downloaded from the internet. In this case no digital signature is

required, so you could be opening your machine up to the risk of running unsigned and potentially malicious scripts downloaded from the internet.

In this tutorial you will use the `RemoteSigned` execution policy to set the permissions for the current user. This will allow the PowerShell to accept trusted scripts without making the permissions as broad as they would be with an `Unrestricted` permission. Enter the following in PowerShell:

```
Set-ExecutionPolicy -Scope CurrentUser
```

PowerShell will then prompt you to provide an execution policy. Enter the following to use `RemoteSigned`:

```
RemoteSigned
```

Once you press `ENTER`, you'll be asked to confirm the change to the execution policy. Type the letter `y` to allow the changes to take effect. You can confirm that this worked by asking for the current permissions across the machine:

```
Get-ExecutionPolicy -List
```

You should receive output that looks something like this:

Output

```
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser    RemoteSigned
 LocalMachine       Undefined
```

This confirms that the current user can run trusted scripts downloaded from the internet. You can now move on to downloading the files we will need to set up our Go programming environment.

Step 2 — Installing the Package Manager Chocolatey

A package manager is a collection of software tools that work to automate installation processes. This includes the initial installation, upgrading and configuring of software, and removing software as needed. They keep software installations in a central location and can maintain all software packages on the system in formats that are commonly used.

Chocolatey is a command-line package manager built for Windows that works like `apt-get` does on Linux. Available in an open-source version, Chocolatey will help you quickly install applications and tools. You will be using it to

download what you need for your development environment.

Before installing the script, read it to confirm that you are happy with the changes it will make to your machine. To do this, use the .NET scripting framework to download and display the Chocolatey script within the terminal window.

Start by creating a WebClient object called `$script` that shares internet connection settings with Internet Explorer:

```
$script = New-Object Net.WebClient
```

Take a look at the available options by piping the `$script` object with `|` to the `Get-Member` class:

```
$script | Get-Member
```

This will return all members (properties and methods) of this WebClient object:

```
 . . .
[secondary_label Snippet of Output]
DownloadFileAsync         Method     void
DownloadFileAsync(uri address, string fileName),
void DownloadFileAsync(ur...
DownloadFileTaskAsync     Method
System.Threading.Tasks.Task
DownloadFileTaskAsync(string address, string
fileNa...
DownloadString            Method     string
DownloadString(string address), string
DownloadString(uri address) #method we will use
```

```
DownloadStringAsync       Method     void
DownloadStringAsync(uri address), void
DownloadStringAsync(uri address, Sy...
DownloadStringTaskAsync   Method
System.Threading.Tasks.Task[string]
DownloadStringTaskAsync(string address), Sy…
 . . .
```

Looking over the output, you can identify the `DownloadString` method used to display the script and signature in the PowerShell window. Use this method to inspect the script:

```
$script.DownloadString("https://chocolatey.org/install.ps1")
```

After inspecting the script, install Chocolatey by typing the following into PowerShell:

```
iwr https://chocolatey.org/install.ps1 -UseBasicParsing | iex
```

The cmdlet `iwr`, or `Invoke-WebRequest`, allows you to extract data from the web. This will pass the script to `iex`, or the `Invoke-Expression` cmdlet, which will execute the contents of the script and run the installation for the Chocolatey package manager.

Allow PowerShell to install Chocolatey. Once it is fully installed, you can begin installing additional tools with the `choco` command.

If you need to upgrade Chocolatey at any time in the future, run the following command:

```
choco upgrade chocolatey
```

With the package manager installed, you can install the rest of what you need for the Go programming environment.

Step 3 — Installing the Text Editor Nano (Optional)

In this step, you are going to install nano, a text editor that uses a command-line interface. You can use nano to write programs directly within PowerShell. This is not a compulsory step, as you can also use a text editor with a graphical user interface such as Notepad. This tutorial recommends using nano, as it will help accustom you to using PowerShell.

Use Chocolatey to install nano:

```
choco install -y nano
```

The `-y` flag automatically confirms that you want to run the script without being prompted for confirmation.

Once nano is installed, you can use the `nano` command to create new text files. You will use it later in this tutorial to write your first Go program.

Step 4 — Installing Go

Just like you did with nano in the previous step, you will use Chocolatey to install Go:

```
choco install -y golang
```

Note: Because go is such a small word, it has become common to use `golang` as a term for installing packages and when searching the internet for Go-related articles. The term Golang was born from the domain for Go, which is `golang.org`.

PowerShell will now install Go, generating output within PowerShell during that process. Once the install is completed, you should see the following output:

Output

```
Environment Vars (like PATH) have changed.
Close/reopen your shell to
see the changes (or in powershell/cmd.exe just
type `refreshenv`).
The install of golang was successful.
 Software installed as 'msi', install location is
likely default.

Chocolatey installed 1/1 packages.
See the log for details
(C:\ProgramData\chocolatey\logs\chocolatey.log).
```

With the installation finished, you'll now confirm that Go is installed. To see the changes, close and re-open PowerShell as an Administrator, then check the version of Go available on your local machine:

```
go version
```

You'll receive output similar to the following:

Output

```
go version go1.12.1 windows/amd643.7.0
```

Once Go is installed, you can set up a workspace for your development projects.

Step 5 — Creating Your Go Workspace

Now that you have Chocolatey, nano, and Go installed, you can create your programming workspace.

The Go workspace will contain two directories at its root:

- `src`: The directory that contains Go source files. A source file is a file that you write using the Go programming language. Source files are used by the Go compiler to create an executable binary file.
- `bin`: The directory that contains executables built and installed by the Go tools. Executables are binary files that run on your system and execute tasks. These are typically the programs compiled by your source code or another downloaded Go source code.

The `src` subdirectory may contain multiple version control repositories (such as [Git](), [Mercurial](), and [Bazaar]()). You will see directories like `github.com` or `golang.org` when your program imports third party libraries. If you are

using a code repository like `github.com`, you will also put your projects and source files under that directory. This allows for a canonical import of code in your project. Canonical imports are imports that reference a fully qualified package, such as `github.com/digitalocean/godo`.

Here is what a typical workspace may look like:

```
.
├── bin
│   ├── buffalo
# command executable
│   ├── dlv
# command executable
│   └── packr
# command executable
└── src
    └── github.com
        └── digitalocean
            └── godo
                ├── .git
# Git repository metadata
                ├── account.go
# package source
                ├── account_test.go
# test source
                ├── ...
                ├── timestamp.go
```

```
├── timestamp_test.go
└── util
    ├── droplet.go
    └── droplet_test.go
```

The default directory for the Go workspace as of 1.8 is your user's home directory with a `go` subdirectory, or `$HOME/go`. If you are using an earlier version of Go than 1.8, it is still considered best practice to use the `$HOME/go` location for your workspace

Issue the following command to navigate to the `$HOME` directory:

```
cd $HOME
```

Next, create the directory structure for your Go workspace:

```
mkdir go/bin, go/src
```

This will ensure the following directory structure is now in place:

```
└── $HOME
    └── go
        ├── bin
        └── src
```

Prior to Go 1.8, it was required to set a local environment variable called `$GOPATH`. While it is no longer explicitly required to do so, it is still considered a good practice as many third party tools still depend on this variable being set.

Since you used Chocolatey for the installation, this environment variable should already be set. You can verify this with the following command:

```
$env:GOPATH
```

You should see the following output, with your username in place of `sammy`:

Output

```
C:\Users\sammy\go
```

When Go compiles and installs tools, it will put them in the `$GOPATH/bin` directory. For convenience, it's common to add the workspace's `bin` subdirectory to your `$PATH`. You can do this using the `setx` command in PowerShell:

```
setx PATH "$($env:path);$GOPATH\bin"
```

This will now allow you to run any programs you compile or download via the Go tools anywhere on your system.

Now that you have the root of the workspace created and your `$GOPATH` environment variable set, you will create your future projects with the following directory structure. This example assumes you are using github.com as your repository:

```
$GOPATH/src/github.com/username/project
```

If you were working on the https://github.com/digitalocean/godo project, you would put it in the following directory:

```
$GOPATH/src/github.com/digitalocean/godo
```

Structuring your projects in this manner will make projects available with the `go get` tool. It will also help readability later.

You can verify this by using the `go get` command to fetch the `godo` library:

```
go get github.com/digitalocean/godo
```

Note: If you don't have `git` installed, Windows will open a dialog box asking if you want to install it. Click Yes to continue and follow the installation instructions.

You can see it successfully downloaded the `godo` package by listing the directory:

```
ls $env:GOPATH/src/github.com/digitalocean/godo
```

You will receive output similar to this:

Output

```
    Directory:
C:\Users\sammy\go\src\github.com\digitalocean\godo


Mode                LastWriteTime         Length
Name
----                -------------         ------ -
---
d-----        4/10/2019   2:59 PM
util
-a----        4/10/2019   2:59 PM              9
.gitignore
-a----        4/10/2019   2:59 PM             69
.travis.yml
-a----        4/10/2019   2:59 PM           1592
account.go
-a----        4/10/2019   2:59 PM           1679
account_test.go
-rw-r--r--  1 sammy  staff   2892 Apr  5 15:56
CHANGELOG.md
-rw-r--r--  1 sammy  staff   1851 Apr  5 15:56
CONTRIBUTING.md
.
.
.
-a----        4/10/2019   2:59 PM           5076
```

```
vpcs.go
-a----        4/10/2019   2:59 PM           4309
vpcs_test.go
```

In this step, you created a Go workspace and configured the necessary environment variables. In the next step you will test the workspace with some code.

Step 6 — Creating a Simple Program

Now that you have the Go workspace set up, create a simple “Hello, World!” program. This will make sure that your workspace is configured properly, and also gives you the opportunity to become more familiar with Go. Because you are creating a single Go source file, and not an actual project, you don’t need to be in your workspace to do this.

From your home directory, open up a command-line text editor, such as nano, and create a new file:

```
nano hello.go
```

Once the text file opens up in nano, type out your program:

hello.go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Exit `nano` by pressing the `CTRL` and `X` keys. When prompted to save the file, press `Y` and then `ENTER`.

This code will use the `fmt` package and call the `Println` function with `Hello, World!` as the argument. This will cause the phrase `Hello, World!` to print out to the terminal when the program is run.

Once you exit out of `nano` and return to your shell, run the program:

```
go run hello.go
```

The hello.go program that you just created should cause PowerShell to produce the following output:

Output

```
Hello, World!
```

In this step, you used a basic program to verify that your Go workspace is properly configured.

Conclusion

Congratulations! At this point you have a Go programming workspace set up on your local Windows machine and can begin a coding project!

How To Write Your First Program in Go

Written by Gopher Guides

The “Hello, World!” program is a classic and time-honored tradition in computer programming. It’s a simple and complete first program for beginners, and it’s a good way to make sure your environment is properly configured.

This tutorial will walk you through creating this program in Go. However, to make the program more interesting, you’ll modify the traditional “Hello, World!” program so that it asks the user for their name. You’ll then use the name in the greeting. When you’re done with the tutorial, you’ll have a program that looks like this when you run it:

Output

```
Please enter your name.
Sammy
Hello, Sammy! I'm Go!
```

Prerequisites

Before you begin this tutorial, you will need a local Go development environment set up on your computer. You can set this up by following one of these tutorials:

- How to Install Go and Set Up a Local Programming Environment on macOS
- How to Install Go and Set Up a Local Programming Environment on Ubuntu 18.04
- How to Install Go and Set Up a Local Programming Environment on Windows 10

Step 1 — Writing the Basic "Hello, World!" Program

To write the "Hello, World!" program, open up a command-line text editor such as `nano` and create a new file:

```
nano hello.go
```

Once the text file opens up in the terminal window, you'll type out your program:

hello.go

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello, World!")
}
```

Let's break down the different components of the code.

`package` is a Go keyword that defines which code bundle this file belongs to. There can be only one package per

folder, and each `.go` file has to declare the same package name at the top of its file. In this example, the code belongs to the `main` package.

`import` is a Go keyword that tells the Go compiler which other packages you want to use in this file. Here you import the `fmt` package that comes with the standard library. The `fmt` package provides formatting and printing functions that can be useful when developing.

`fmt.Println` is a Go function, found in the `fmt` package, that tells the computer to print some text to the screen.

You follow the `fmt.Println` function by a sequence of characters, like `"Hello, World!"`, enclosed in quotation marks. Any characters that are inside of quotation marks are called a string. The `fmt.Println` function will print this string to the screen when the program runs.

Save and exit `nano` by typing `CTRL + X`, when prompted to save the file, press `Y`.

Now you can try your program.

Step 2 — Running a Go Program

With your "Hello, World!" program written, you're ready to run the program. You'll use the `go` command, followed by the name of the file you just created.

```
go run hello.go
```

The program will execute and display this output:

Output

```
Hello, World!
```

Let's explore what actually happened.

Go programs need to compile before they run. When you call `go run` with the name of a file, in this case `hello.go`, the `go` command will compile the application and then run the resulting binary. For programs written in compiled programming languages, a compiler will take the source code of a program and generate another type of lower-level code (such as machine code) to produce an executable program.

Go applications require a `main` package and exactly one `main()` function that serves as the entry point for the application. The `main` function takes no arguments and returns no values. Instead it tells the Go compiler that the package should be compiled as an executable package.

Once compiled, the code executes by entering the `main()` function in the `main` package. It executes the line `fmt.Println("Hello, World!")` by calling the `fmt.Println` function. The string value of `Hello, World!` is then passed to the function. In this example, the string `Hello, World!` is also called an argument since it is a value that is passed to a method.

The quotes that are on either side of `Hello, World!` are not printed to the screen because you use them to tell Go where your string begins and ends.

In this step, you've created a working "Hello, World!" program with Go. In the next step, you will explore how to make the program more interactive.

Step 3 — Prompting for User Input

Every time you run your program, it produces the same output. In this step, you can add to your program to prompt the user for their name. You'll then use their name in the output.

Instead of modifying your existing program, create a new program called `greeting.go` with the `nano` editor:

```
nano greeting.go
```

First, add this code, which prompts the user to enter their name:

greeting.go

```go
package main

import (
    "fmt"
)

func main() {
  fmt.Println("Please enter your name.")
}
```

Once again, you use the `fmt.Println` function to print some text to the screen.

Now add the highlighted line to store the user's input:

greeting.go

```
package main

import (
    "fmt"
)

func main() {
  fmt.Println("Please enter your name.")
  var name string
}
```

The `var name string` line will create a new variable using the `var` keyword. You name the variable `name`, and it will be of type `string`.

Then, add the highlighted line to capture the user's input:

greeting.go

```
package main

import (
    "fmt"
)

func main() {
  fmt.Println("Please enter your name.")
  var name string
  fmt.Scanln(&name)
}
```

The `fmt.Scanln` method tells the computer to wait for input from the keyboard ending with a new line or (`\n`), character. This pauses the program, allowing the user to enter any text they want. The program will continue when the user presses the `ENTER` key on their keyboard. All of the keystrokes, including the `ENTER` keystroke, are then captured and converted to a string of characters.

You want to use those characters in your program's output, so you save those characters by writing them into the string variable called `name`. Go stores that string in your computer's memory until the program finishes running.

Finally, add the following highlighted line in your program to print the output:

greeting.go

```
package main

import (
    "fmt"
)

func main() {
  fmt.Println("Please enter your name.")
  var name string
  fmt.Scanln(&name)
  fmt.Printf("Hi, %s! I'm Go!", name)
}
```

This time, instead of using the `fmt.Println` method again, you're using `fmt.Printf`. The `fmt.Printf` function takes a string, and using special printing verbs, (`%s`), it injects the value of `name` into the string. You do this because Go does not support string interpolation, which would let you take the value assigned to a variable and place it inside of a string.

Save and exit `nano` by pressing `CTRL + X`, and press `Y` when prompted to save the file.

Now run the program. You'll be prompted for your name, so enter it and press `ENTER`. The output might not be exactly what you expect:

Output

```
Please enter your name.
Sammy
Hi, Sammy
! I'm Go!
```

Instead of `Hi, Sammy! I'm Go!`, there's a line break right after the name.

The program captured all of our keystrokes, including the `ENTER` key that we pressed to tell the program to continue. In a string, pressing the `ENTER` key creates a special character that creates a new line. The program's output is doing exactly what you told it to do; it's displaying the text you entered, including that new line. It's not what you expected the output to be, but you can fix it with additional functions.

Open the `greeting.go` file in your editor:

```
nano greeting.go
```

Locate this line in your program:

greeting.go

```
...
fmt.Scanln(&name)
...
```

Add the following line right after it:

greeting.go

```
name = strings.TrimSpace(name)
```

This uses the `TrimSpace` function, from Go's standard library `strings` package, on the string that you captured with `fmt.Scanln`. The `strings.TrimSpace` function removes any space characters, including new lines, from the start and end of a string. In this case, it removes the newline character at the end of the string created when you pressed `ENTER`.

To use the `strings` package you need to import it at the top of the program.

Locate these lines in your program:

greeting.go

```
import (
    "fmt"
)
```

Add the following line to import the `strings` package:

greeting.go

```
import (
    "fmt"
  "strings"
)
```

Your program will now contain the following:

greeting.go

```
package main

import (
	"fmt"
	"strings"
)

func main() {
    fmt.Println("Please enter your name.")
    var name string
    fmt.Scanln(&name)
    fmt.Printf("Hi, %s! I'm Go!", name)
    name = strings.TrimSpace(name)
}
```

Save and exit `nano`. Press `CTRL + X`, then press `Y` when prompted to save the file.

Run the program again:

```
go run greeting.go
```

This time, after you enter your name and press `ENTER`, you get the expected output:

Output

```
Please enter your name.
Sammy
Hi, Sammy! I'm Go!
```

You now have a Go program that takes input from a user and prints it back to the screen.

Conclusion

In this tutorial, you wrote a “Hello, World!” program that takes input from a user, processes the results, and displays the output. Now that you have a basic program to work with, try to expand your program further. For example, ask for the user’s favorite color, and have the program say that its favorite color is red. You might even try to use this same technique to create a simple Mad-Lib program.

Understanding the GOPATH

Written by Gopher Guides

This article will walk you through understanding what the `GOPATH` is, how it works, and how to set it up. This is a crucial step for setting up a Go development environment, as well as understanding how Go finds, installs, and builds source files. In this article we will use `GOPATH` when referring to the concept of the folder structure we will be discussing. We will use `$GOPATH` to refer to the environment variable that Go uses to find the folder structure.

A Go Workspace is how Go manages our source files, compiled binaries, and cached objects used for faster compilation later. It is typical, and also advised, to have only one Go Workspace, though it is possible to have multiple spaces. The `GOPATH` acts as the root folder of a workspace.

Setting the $GOPATH Environment Variable

The `$GOPATH` environment variable lists places for Go to look for Go Workspaces.

By default, Go assumes our `GOPATH` location is at `$HOME/go`, where `$HOME` is the root directory of our user account on our computer. We can change this by setting the `$GOPATH` environment variable. For further study, follow this

tutorial on reading and setting environment variables in Linux.

For more information on setting the `$GOPATH` variable, see the Go documentation.

Furthermore, this series walks through installing Go and setting up a Go development environment.

$GOPATH Is Not $GOROOT

The `$GOROOT` is where Go's code, compiler, and tooling lives — this is not our source code. The `$GOROOT` is usually something like `/usr/local/go`. Our `$GOPATH` is usually something like `$HOME/go`.

While we don't need to specifically set up the `$GOROOT` variable anymore, it is still referenced in older materials.

Now, let's discuss the structure of the Go Workspace.

Anatomy of the Go Workspace

Inside of a Go Workspace, or `GOPATH`, there are three directories: `bin`, `pkg`, and `src`. Each of these directories has special meaning to the Go tool chain.

```
.
├── bin
├── pkg
└── src
  └── github.com/foo/bar
    └── bar.go
```

Let's take a look at each of these directories.

The `$GOPATH/bin` directory is where Go places binaries that `go install` compiles. Our operating system uses the `$PATH` environment variable to find binary applications that can execute without a full path. It is recommended to add this directory to our global `$PATH` variable.

For example, if we don't add `$GOPATH/bin` to `$PATH` to execute a program from there, we would need to run:

```
$GOPATH/bin/myapp
```

When `$GOPATH/bin` is added to `$PATH` we can make the same call like such:

```
myapp
```

The `$GOPATH/pkg` directory is where Go stores pre-compiled object files to speed up subsequent compiling of programs. Typically, most developers won't need to access this directory. If you experience issues with compilation, you can safely delete this directory and Go will then rebuild it.

The `src` directory is where all of our `.go` files, or source code, must be located. This shouldn't be confused with the source code the Go tooling uses, which is located at the `$GOROOT`. As we write Go applications, packages, and libraries, we will place these files under `$GOPATH/src/path/to/code`.

What Are Packages?

Go code is organized in packages. A package represents all the files in a single directory on disk. One directory can contain only certain files from the same package. Packages are stored, with all user-written Go source files, under the `$GOPATH/src` directory. We can understand package resolution by importing different packages.

If our code lives at `$GOPATH/src/blue/red` then its package name should be `red`.

The import statement for the `red` package would be:

```
import "blue/red"
```

Packages that live in source code repositories, like GitHub and BitBucket, have the full location of the repository as part of their import path.

For example, we would import the source code at https://github.com/gobuffalo/buffalo using the following import path:

```
import "github.com/gobuffalo/buffalo"
```

Therefore, this source code would be in the following location on disk:

```
$GOPATH/src/github.com/gobuffalo/buffalo
```

Conclusion

In this article we discussed the `GOPATH` as a set of folder's that Go expects our source code to live within, as well as what those folders are, and what they contain. We discussed how to change that location from the default of

`$HOME/go` to the user's choice by setting the `$GOPATH` environment variable. Finally, we discussed how Go searches for packages within that folder structure.

Introduced in Go 1.11, Go Modules aim to replace Go Workspaces and the `GOPATH`. While it is recommended to start using modules, some environments, such as corporate environments, may not be ready to use modules.

The `GOPATH` is one of the trickier aspects of setting up Go, but once it is set up, we can usually forget about it.

How To Write Comments in Go

Written by Gopher Guides

Comments are lines that exist in computer programs that are ignored by compilers and interpreters. Including comments in programs makes code more readable for humans as it provides some information or explanation about what each part of a program is doing.

Depending on the purpose of your program, comments can serve as notes to yourself or reminders, or they can be written with the intention of other programmers being able to understand what your code is doing.

In general, it is a good idea to write comments while you are writing or updating a program as it is easy to forget your thought process later on, and comments written later may be less useful in the long term.

Comment Syntax

Comments in Go begin with a set of forward slashes (`//`) and continue to the end of the line. It is idiomatic to have a white space after the set of forward slashes.

Generally, comments will look something like this:

```
// This is a comment
```

Comments do not execute, so there will be no indication of a comment when running a program. Comments are in the source code for humans to read, not for computers to execute.

In a “Hello, World!” program, a comment may look like this:

hello.go

```
package main

import (
    "fmt"
)

func main() {
    // Print “Hello, World!” to console
    fmt.Println("Hello, World!")
}
```

In a `for` loop that iterates over a slice, comments may look like this:

sharks.go

```
package main

import (
    "fmt"
)

func main() {
    // Define sharks variable as a slice of strings
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled", "bu

    // For loop that iterates over sharks list and prints each string item
    for _, shark := range sharks {
        fmt.Println(shark)
    }
}
```

Comments should be made at the same indent as the code it is commenting. That is, a function definition with no indent would have a comment with no indent, and each indent level following would have comments that are aligned with the code it is commenting.

For example, here is how the `main` function is commented, with comments following each indent level of the code:

color.go

```
package main

import "fmt"

const favColor string = "blue"

func main() {
    var guess string
    // Create an input loop
    for {
        // Ask the user to guess my favorite color
        fmt.Println("Guess my favorite color:")
        // Try to read a line of input from the user. Print out the error 0
        if _, err := fmt.Scanln(&guess); err != nil {
            fmt.Printf("%s\n", err)
            return
        }
        // Did they guess the correct color?
        if favColor == guess {
            // They guessed it!
            fmt.Printf("%q is my favorite color!\n", favColor)
            return
        }
        // Wrong! Have them guess again.
        fmt.Printf("Sorry, %q is not my favorite color. Guess again.\n", gues
    }
}
```

Comments are made to help programmers, whether it is the original programmer or someone else using or collaborating on the project. If comments cannot be properly maintained and updated along with the code base, it is better to not include a comment rather than write a comment that contradicts or will contradict the code.

When commenting code, you should be looking to answer the why behind the code as opposed to the what or how. Unless the code is particularly tricky, looking at the code can generally answer the what or how, which is why comments are usually focused around the why.

Block Comments

Block comments can be used to explain more complicated code or code that you don't expect the reader to be familiar with.

You can create block comments two ways in Go. The first is by using a set of double forward slashes and repeating them for every line.

```
// First line of a block comment
// Second line of a block comment
```

The second is to use opening tags (`/*`) and closing tags (`*/`). For documenting code, it is considered idiomatic to always use `//` syntax. You would only use the `/* ... */` syntax for debugging, which we will cover later in this article.

```
/*
Everything here
will be considered
a block comment
*/
```

In this example, the block comment defines what is happening in the `MustGet()` function:

function.go

```go
// MustGet will retrieve a url and return the body of the page.
// If Get encounters any errors, it will panic.
func MustGet(url string) string {
	resp, err := http.Get(url)
	if err != nil {
		panic(err)
	}

	// don't forget to close the body
	defer resp.Body.Close()
	var body []byte
	if body, err = ioutil.ReadAll(resp.Body); err != nil {
		panic(err)
	}
	return string(body)
}
```

It is common to see block comments at the beginning of exported functions in Go; these comments are also what generate your code documentation. Block comments are also used when operations are less straightforward and are therefore demanding of a thorough explanation. With the exception of documenting functions, you should try to avoid over-

commenting the code and trust other programmers to understand Go, unless you are writing for a particular audience.

Inline Comments

Inline comments occur on the same line of a statement, following the code itself. Like other comments, they begin with a set of forward slashes. Again, it's not required to have a whitespace after the forward slashes, but it is considered idiomatic to do so.

Generally, inline comments look like this:

```
[code]  // Inline comment about the code
```

Inline comments should be used sparingly, but can be effective for explaining tricky or non-obvious parts of code. They can also be useful if you think you may not remember a line of the code you are writing in the future, or if you are collaborating with someone who you know may not be familiar with all aspects of the code.

For example, if you don't use a lot of math in your Go programs, you or your collaborators may not know that the following creates a complex number, so you may want to include an inline comment about that:

```
z := x % 2  // Get the modulus of x
```

You can also use inline comments to explain the reason behind doing something, or to provide some extra information, as in:

```
x := 8  // Initialize x with an arbitrary number
```

You should only use inline comments when necessary and when they can provide helpful guidance for the person reading the program.

Commenting Out Code for Testing

In addition to using comments as a way to document code, you can also use opening tags (`/*`) and closing tags (`*/`) to create a block comment. This allows you to comment out code that you don't want to execute while you are testing or debugging a program you are currently creating. That is, when you experience errors after implementing new lines of code, you may want to comment a few of them out to see if you can troubleshoot the precise issue.

Using the `/*` and `*/` tags can also allow you to try alternatives while you're determining how to set up your code. You can also use block comments to comment out code that is failing while you continue to work on other parts of your code.

multiply.go

```go
// Function to add two numbers
func addTwoNumbers(x, y int) int {
    sum := x + y
    return sum
}

// Function to multiply two numbers
func multiplyTwoNumbers(x, y int) int {
    product := x * y
    return product
}

func main() {
    /*
        In this example, we're commenting out the addTwoNumbers
        function because it is failing, therefore preventing it from executin
        Only the multiplyTwoNumbers function will run

        a := addTwoNumbers(3, 5)
        fmt.Println(a)

    */

    m := multiplyTwoNumbers(5, 9)
    fmt.Println(m)
}
```

Note: Commenting out code should only be done for testing purposes. Do not leave snippets of commented out code in your final program.

Commenting out code with the `/*` and `*/` tags can allow you to try out different programming methods as well as help you find the source of an error through systematically commenting out and running parts of a program.

Conclusion

Using comments within your Go programs helps to make your programs more readable for humans, including your future self. Adding appropriate comments that are relevant and useful can make it easier for others to collaborate with you on programming projects and make the value of your code more obvious.

Commenting your code properly in Go will also allow for you to use the Godoc tool. Godoc is a tool that will extract comments from your code and generate documentation for your Go program.

Understanding Data Types in Go

Written by Gopher Guides

Data types specify the kinds of values that particular variables will store when you are writing a program. The data type also determines what operations can be performed on the data.

In this article, we will go over the important data types native to Go. This is not an exhaustive investigation of data types, but will help you become familiar with what options you have available to you in Go. Understanding some basic data types will enable you to write clearer code that performs efficiently.

Background

One way to think about data types is to consider the different types of data that we use in the real world. An example of data in the real world are numbers: we may use whole numbers (0, 1, 2, …), integers (…, -1, 0, 1, …), and irrational numbers (π), for example.

Usually, in math, we can combine numbers from different types, and get some kind of an answer. We may want to add 5 to π, for example:

```
5 + π
```

We can either keep the equation as the answer to account for the irrational number, or round π to a number with an abbreviated number of decimal places, and then add the numbers together:

```
5 + π = 5 + 3.14 = 8.14
```

But, if we start to try to evaluate numbers with another data type, such as words, things start to make less sense. How would we solve for the following equation?

```
shark + 8
```

For computers, each data type is quite different—like words and numbers. As a result we have to be careful about how we use varying data types to assign values and how we manipulate them through operations.

Integers

Like in math, integers in computer programming are whole numbers that can be positive, negative, or 0 (…, -1, 0, 1, …). In Go, an integer is known as an `int`. As with other programming languages, you should not use commas in numbers of four digits or more, so when you write 1,000 in your program, write it as `1000`.

We can print out an integer in a simple way like this:

```
fmt.Println(-459)
```

Output

```
-459
```

Or, we can declare a variable, which in this case is a symbol of the number we are using or manipulating, like so:

```
var absoluteZero int = -459
fmt.Println(absoluteZero)
```

Output

```
-459
```

We can do math with integers in Go, too. In the following code block, we will use the `:=` assignment operator to declare and instantiate the variable `sum`:

```
sum := 116 - 68
fmt.Println(sum)
```

Output

```
48
```

As the output shows, the mathematical operator `-` subtracted the integer `68` from `116`, resulting in `48`. You'll learn more about variable declaration in the Declaring Data Types for Variables section.

Integers can be used in many ways within Go programs. As you continue to learn about Go, you'll have a lot of opportunities to work with integers and build upon your knowledge of this data type.

Floating-Point Numbers

A floating-point number or a float is used to represent real numbers that cannot be expressed as integers. Real numbers include all rational and irrational numbers, and because of this, floating-point numbers can contain a fractional part, such as 9.0 or -116.42. For the purposes of thinking of a float in a Go program, it is a number that contains a decimal point.

Like we did with integers, we can print out a floating-point number in a simple way like this:

```
fmt.Println(-459.67)
```

Output

```
-459.67
```

We can also declare a variable that stands in for a float, like so:

```
absoluteZero := -459.67
fmt.Println(absoluteZero)
```

Output

```
-459.67
```

Just like with integers, we can do math with floats in Go, too:

```
var sum = 564.0 + 365.24
fmt.Println(sum)
```

Output

```
929.24
```

With integers and floating-point numbers, it is important to keep in mind that 3 ≠ 3.0, as 3 refers to an integer while 3.0 refers to a float.

Sizes of Numeric Types

In addition to the distinction between integers and floats, Go has two types of numeric data that are distinguished by the static or dynamic nature of their sizes. The first type is an architecture-independent type, which means that the size of the data in bits does not change, regardless of the machine that the code is running on.

Most system architectures today are either 32 bit or 64 bit. For instance, you may be developing for a modern Windows laptop, on which the operating system runs on a 64-bit architecture. However, if you are developing for a device like a fitness watch, you may be working with a 32-bit architecture. If you use an architecture-independent type like `int32`, regardless of the architecture you compile for, the type will have a constant size.

The second type is an implementation-specific type. In this type, the bit size can vary based on the architecture the program is built on. For instance, if we use the `int` type, when Go compiles for a 32-bit architecture, the size of the data type will be 32 bits. If the program is compiled for a 64-bit architecture, the variable will be 64 bits in size.

In addition to data types having different sizes, types like integers also come in two basic types: signed and unsigned. An `int8` is a signed integer, and can have a value from -128 to 127. A `uint8` is an unsigned integer, and can only have a positive value of 0 to 255.

The ranges are based on the bit size. For binary data, 8 bits can represent a total of 256 different values. Because an `int` type needs to support both positive and negative values, an 8-bit integer (`int8`) will have a range of -128 to 127, for a total of 256 unique possible values.

Go has the following architecture-independent integer types:

```
uint8       unsigned  8-bit integers (0 to 255)
uint16      unsigned 16-bit integers (0 to 65535)
uint32      unsigned 32-bit integers (0 to 4294967295)
uint64      unsigned 64-bit integers (0 to 18446744073709551615)
```

```
int8        signed  8-bit integers (-128 to 127)
int16       signed 16-bit integers (-32768 to 32767)
int32       signed 32-bit integers (-2147483648 to 2147483647)
int64       signed 64-bit integers (-9223372036854775808 to
9223372036854775807)
```

Floats and complex numbers also come in varying sizes:

```
float32     IEEE-754 32-bit floating-point numbers
float64     IEEE-754 64-bit floating-point numbers
complex64   complex numbers with float32 real and imaginary parts
complex128  complex numbers with float64 real and imaginary parts
```

There are also a couple of alias number types, which assign useful names to specific data types:

```
byte        alias for uint8
rune        alias for int32
```

The purpose of the `byte` alias is to make it clear when your program is using bytes as a common computing measurement in character string elements, as opposed to small integers unrelated to the byte data measurement. Even though `byte` and `uint8` are identical once the program is compiled, `byte` is often used to represent character data in numeric form, whereas `uint8` is intended to be a number in your program.

The `rune` alias is a bit different. Where `byte` and `uint8` are exactly the same data, a `rune` can be a single byte or four bytes, a range determined by `int32`. A `rune` is used to represent a Unicode character, whereas only ASCII characters can be represented solely by an `int32` data type.

In addition, Go has the following implementation-specific types:

```
uint     unsigned, either 32 or 64 bits
int      signed, either 32 or 64 bits
uintptr  unsigned integer large enough to store the uninterpreted bits of a
pointer value
```

Implementation-specific types will have their size defined by the architecture the program is compiled for.

Picking Numeric Data Types

Picking the correct size usually has more to do with performance for the target architecture you are programming for than the size of the data you are working with. However, without needing to know the specific ramifications of performance for your program, you can follow some of these basic guidelines when first starting out.

As discussed earlier in this article, there are architecture-independent types, and implementation-specific types. For integer data, it’s common in Go to use the implementation types like `int` or `uint` instead of `int64` or `uint64`. This will typically result in the fastest processing speed for your target architecture. For instance, if you use

an `int64` and compile to a 32-bit architecture, it will take at least twice as much time to process those values as it takes additional CPU cycles to move the data across the architecture. If you used an `int` instead, the program would define it as a 32-bit size for a 32-bit architecture, and would be significantly faster to process.

If you know you won't exceed a specific size range, then picking an architecture-independent type can both increase speed and decrease memory usage. For example, if you know your data won't exceed the value of `100`, and will only be a positive number, then choosing a `uint8` would make your program more efficient as it will require less memory.

Now that we have looked at some of the possible ranges for numeric data types, let's look at what will happen if we exceed those ranges in our program.

Overflow vs. Wraparound

Go has the potential to both overflow a number and wraparound a number when you try to store a value larger than the data type was designed to store, depending on if the value is calculated at compile time or at runtime. A compile time error happens when the program finds an error as it tries to build the program. A runtime error happens after the program is compiled, while it is actually executing.

In the following example, we set `maxUint32` to its maximum value:

```
package main

import "fmt"

func main() {
    var maxUint32 uint32 = 4294967295 // Max uint32 size
    fmt.Println(maxUint32)
}
```

It will compile and run with the following result:

Output

```
4294967295
```

If we add `1` to the value at runtime, it will wraparound to `0`:

Output

```
0
```

On the other hand, let's change the program to add `1` to the variable when we assign it, before compile time:

```go
package main

import "fmt"

func main() {
	var maxUint32 uint32 = 4294967295 + 1
	fmt.Println(maxUint32)

}
```

At compile time, if the compiler can determine a value will be too large to hold in the data type specified, it will throw an `overflow` error. This means that the value calculated is too large for the data type you specified.

Because the compiler can determine it will overflow the value, it will now throw an error:

Output

```
prog.go:6:36: constant 4294967296 overflows uint32
```

Understanding the boundaries of your data will help you avoid potential bugs in your program in the future.

Now that we have covered numeric types, let's look at how to store boolean values.

Booleans

The boolean data type can be one of two values, either `true` or `false`, and is defined as `bool` when declaring it as a data type. Booleans are used to represent the truth values that are associated with the logic branch of mathematics, which informs algorithms in computer science.

The values `true` and `false` will always be with a lowercase `t` and `f` respectively, as they are pre-declared identifiers in Go.

Many operations in math give us answers that evaluate to either true or false:

- greater than
    - 500 > 100 true
    - 1 > 5 false
- less than
    - 200 < 400 true
    - 4 < 2 false
- equal

- 5 = 5 true
- 500 = 400 false

Like with numbers, we can store a boolean value in a variable:

```
myBool := 5 > 8
```

We can then print the boolean value with a call to the `fmt.Println()` function:

```
fmt.Println(myBool)
```

Since `5` is not greater than `8`, we will receive the following output:

Output

```
false
```

As you write more programs in Go, you will become more familiar with how booleans work and how different functions and operations evaluating to either `true` or `false` can change the course of the program.

Strings

A string is a sequence of one or more characters (letters, numbers, symbols) that can be either a constant or a variable. Strings exist within either back quotes ` or double quotes " in Go and have different characteristics depending on which quotes you use.

If you use the back quotes, you are creating a raw string literal. If you use the double quotes, you are creating an interpreted string literal.

Raw String Literals

Raw string literals are character sequences between back quotes, often called back ticks. Within the quotes, any character will appear just as it is displayed between the back quotes, except for the back quote character itself.

```
a := `Say "hello" to Go!`
fmt.Println(a)
```

Output

```
Say "hello" to Go!
```

Usually, backslashes are used to represent special characters in strings. For example, in an interpreted string, `\n` would represent a new line in a string. However, backslashes have no special meaning inside of raw string literals:

```
a := `Say "hello" to Go!\n`
fmt.Println(a)
```

Because the backslash has no special meaning in a string literal, it will actually print out the value of `\n` instead of making a new line:

Output

```
Say "hello" to Go!\n
```

Raw string literals may also be used to create multiline strings:

```
a := `This string is on
multiple lines
within a single back
quote on either side.`
fmt.Println(a)
```

Output

```
This string is on
multiple lines
within a single back
quote on either side.
```

In the preceding code blocks, the new lines were carried over literally from input to output.

Interpreted String Literals

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear except newline and unescaped double quotes. To show double quotes in an interpreted string, you can use the backslash as an escape character, like so:

```
a := "Say \"hello\" to Go!"
fmt.Println(a)
```

Output

```
Say "hello" to Go!
```

You will almost always use interpreted string literals because they allow for escape characters within them. For more on working with strings, check out [An Introduction to Working with Strings in Go].

Strings with UTF-8 Characters

UTF-8 is an encoding scheme used to encode variable width characters into one to four bytes. Go supports UTF-8 characters out of the box, without any special setup, libaries, or packages. Roman characters such as the letter `A` can be represented by an ASCII value such as the number 65. However, with special characters such as an international character of □, UTF-8 would be required. Go uses the `rune` alias type for UTF-8 data.

```
a := "Hello, □□"
```

You can use the `range` keyword in a `for` loop to index through any string in Go, even a UTF-8 string. `for` loops and `range` will be covered in more depth later in the series; for now, it's important to know that we can use this to count the bytes in a given string:

```
package main

import "fmt"

func main() {
    a := "Hello, □□"
    for i, c := range a {
        fmt.Printf("%d: %s\n", i, string(c))
    }
    fmt.Println("length of 'Hello, □□': ", len(a))
}
```

In the above code block, we declared the variable `a` and assigned the value of `Hello, □□` to it. The text assigned has UTF-8 characters in it.

We then used a standard `for` loop as well as the `range` keyword. In Go, the `range` keyword will index through a string returning one character at a time, as well as the byte index the character is at in the string.

Using the `fmt.Printf` function, we provided a format string of `%d: %s\n`. `%d` is the print verb for a digit (in this case an integer), and `%s` is the print verb for a string. We then provided the values of `i`, which is the current index of the `for` loop, and `c`, which is the current character in the `for` loop.

Finally, we printed the entire length of the variable `a` by using the builtin `len` function.

Earlier, we mentioned that a rune is an alias for `int32` and can be made up of one to four bytes. The □ character takes three bytes to define and the index moves accordingly when ranging through the UTF-8 string. This is the reason that `i` is not sequential when it is printed out.

Output

```
0: H
1: e
2: l
3: l
4: o
5: ,
6:
7: □
10: □
length of 'Hello, □□':  13
```

As you can see, the length is longer than the number of times it took to range over the string.

You won’t always be working with UTF-8 strings, but when you are, you’ll now understand why they are runes and not a single `int32`.

Declaring Data Types for Variables

Now that you know about the different primitive data types, we will go over how to assign these types to variables in Go.

In Go, we can define a variable with the keyword `var` followed by the name of the variable and the data type desired.

In the following example we will declare a variable called `pi` of type `float64`.

The keyword `var` is the first thing declared:

```go
var pi float64
```

Followed by the name of our variable, `pi`:

```go
var pi float64
```

And finally the data type `float64`:

```go
var pi float64
```

We can optionally specify an initial value as well, such as `3.14`:

```go
var pi float64 = 3.14
```

Go is a statically typed language. Statically typed means that each statement in the program is checked at compile time. It also means that the data type is bound to the variable, whereas in dynamically linked languages, the data type is bound to the value.

For example, in Go, the type is declared when declaring a variable:

```go
var pi float64 = 3.14
var week int = 7
```

Each of these variables could be a different data type if you declared them differently.

This is different from a language like PHP, where the data type is associated to the value:

```
$s = "sammy";         // $s is automatically a string
$s = 123;             // $s is automatically an integer
```

In the preceding code block, the first `$s` is a string because it is assigned the value `"sammy"`, and the second is an integer because it has the value `123`.

Next, let's look at more complex data types like arrays.

Arrays

An array is an ordered sequence of elements. The capacity of an array is defined at creation time. Once an array has allocated its size, the size can no longer be changed. Because the size of an array is static, it means that it only allocates memory once. This makes arrays somewhat rigid to work with, but increases performance of your program. Because of this, arrays are typically used when optimizing programs. Slices, covered next, are more flexible, and constitute what you would think of as arrays in other languages.

Arrays are defined by declaring the size of the array, then the data type with the values defined between curly brackets `{ }`.

An array of strings looks like this:

```
[3]string{"blue coral", "staghorn coral", "pillar coral"}
```

We can store an array in a variable and print it out:

```
coral := [3]string{"blue coral", "staghorn coral", "pillar coral"}
fmt.Println(coral)
```

Output

```
[blue coral staghorn coral pillar coral]
```

As mentioned before, slices are similar to arrays, but are much more flexible. Let's take a look at this mutable data type.

Slices

A slice is an ordered sequence of elements that can change in length. Slices can increase their size dynamically. When you add new items to a slice, if the slice does not have enough memory to store the new items, it will request more memory from the system as needed. Because a slice can be expanded to add more elements when needed, they are more commonly used than arrays.

Slices are defined by declaring the data type preceded by an opening and closing square bracket `[]` and having values between curly brackets `{ }`.

A slice of integers looks like this:

```
[]int{-3, -2, -1, 0, 1, 2, 3}
```

A slice of floats looks like this:

```
[]float64{3.14, 9.23, 111.11, 312.12, 1.05}
```

A slice of strings looks like this:

```
[]string{"shark", "cuttlefish", "squid", "mantis shrimp"}
```

Let's define our slice of strings as seaCreatures:

```
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis shrimp"}
```

We can print them out by calling the variable:

```
fmt.Println(seaCreatures)
```

The output will look exactly like the list that we created:

Output

```
[shark cuttlefish squid mantis shrimp]
```

We can use the append keyword to add an item to our slice. The following command will add the string value of seahorse to the slice:

```
seaCreatures = append(seaCreatures, "seahorse")
```

You can verify it was added by printing it out:

```
fmt.Println(seaCreatures)
```

Output

```
[shark cuttlefish squid mantis shrimp seahorse]
```

As you can see, if you need to manage an unknown size of elements, a slice will be much more versatile than an array.

Maps

The map is Go's built-in hash or dictionary type. Maps use keys and values as a pair to store data. This is useful in programming to quickly look up values by an index, or in this case, a key. For instance, you may want to keep a map of users, indexed by their user ID. The key would be the user ID, and the user object would be the value. A map is constructed by using the keyword map followed by the key data type in square brackets [ ], followed by the value data type and the key value pairs in curly braces.

```
map[key]value{}
```

Typically used to hold data that are related, such as the information contained in an ID, a map looks like this:

```
map[string]string{"name": "Sammy", "animal": "shark", "color": "blue", "locat
```

You will notice that in addition to the curly braces, there are also colons throughout the map. The words to the left of the colons are the keys. Keys can be any comparable type in Go. Comparable types are primitive types like strings, ints, etc. A primitive type is defined by the language, and not built from combining any other types. While they can be user-defined types, it's considered best practice to keep them simple to avoid programming errors. The keys in the dictionary above are: name, animal, color, and location.

The words to the right of the colons are the values. Values can be comprised of any data type. The values in the dictionary above are: `Sammy`, `shark`, `blue`, and `ocean`.

Let’s store the map inside a variable and print it out:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark", "color": "blue
fmt.Println(sammy)
```

Output

```
map[animal:shark color:blue location:ocean name:Sammy]
```

If we want to isolate Sammy’s color, we can do so by calling `sammy["color"]`. Let’s print that out:

```
fmt.Println(sammy["color"])
```

Output

```
blue
```

As maps offer key-value pairs for storing data, they can be important elements in your Go program.

Conclusion

At this point, you should have a better understanding of some of the major data types that are available for you to use in Go. Each of these data types will become important as you develop programming projects in the Go language.

Once you have a solid grasp of data types available to you in Go, you can learn [How To Convert Data Types](#) in order to change your data types according to the situation.

An Introduction to Working with Strings in Go

Written by Gopher Guides

A string is a sequence of one or more characters (letters, numbers, symbols) that can be either a constant or a variable. Made up of Unicode, strings are immutable sequences, meaning they are unchanging.

Because text is such a common form of data that we use in everyday life, the string data type is a very important building block of programming.

This Go tutorial will go over how to create and print strings, how to concatenate and replicate strings, and how to store strings in variables.

String Literals

In Go, strings exist within either back quotes ` (sometimes referred to as back ticks) or double quotes ". Depending on which quotes you use, the string will have different characteristics.

Using back quotes, as in ` ```bar``` `, will create a raw string literal. In a raw string literal, any character may appear between quotes, with the exception of back quotes. Here’s an example of a raw string literal:

```
`Say "hello" to Go!`
```

Backslashes have no special meaning inside of raw string literals. For instance, `\n` will appear as the actual characters, backslash `\` and letter `n`. Unlike interpreted string literals, in which `\n` would insert an actual new line.

Raw string literals may also be used to create multi-line strings:

```
`Go is expressive, concise, clean, and efficient.
Its concurrency mechanisms make it easy to write
programs
that get the most out of multi-core and networked
machines,
while its novel type system enables flexible and
modular
program construction. Go compiles quickly to
machine code
yet has the convenience of garbage collection and
the power
of run-time reflection. It's a fast, statically
typed,
compiled language that feels like a dynamically
typed,
interpreted language.`
```

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear with the exception of newline and unescaped double quotes.

```
"Say \"hello\" to Go!"
```

You will almost always use interpreted string literals because they allow for escape characters within them.

Now that you understand how strings are formatted in Go, let's take a look at how you can print strings in programs.

Printing Strings

You can print out strings by using the `fmt` package from the system library and calling the `Println()` function:

```
fmt.Println("Let's print out this string.")
```

Output

```
Let's print out this string.
```

You have to `import` system packages when you use them, so a simple program to print out a string would look like this:

```
package main

import "fmt"

func main() {
    fmt.Println("Let's print out this string.")
}
```

String Concatenation

Concatenation means joining strings together, end-to-end, to create a new string. You can concatenate strings with the + operator. Keep in mind that when you work with numbers, + will be an operator for addition, but when used with strings it is a joining operator.

Let's combine the string literals `"Sammy"` and `"Shark"` together with concatenation through a `fmt.Println()` statement:

```
fmt.Println("Sammy" + "Shark")
```

Output

```
SammyShark
```

If you would like a whitespace between the two strings, you can simply include the whitespace within a string. In this example, add the whitespace within the quotes after `Sammy`:

```
fmt.Println("Sammy " + "Shark")
```

Output

```
Sammy Shark
```

The + operator can not be used between two different data types. As an example, you can't concatenate strings and integers together. If you were to try to write the following:

```
fmt.Println("Sammy" + 27)
```

You will receive the following errors:

Output

```
cannot convert "Sammy" (type untyped string) to
type int
invalid operation: "Sammy" + 27 (mismatched types
string and int)
```

If you wanted to create the string `"Sammy27"`, you could do so by putting the number `27` in quotes (`"27"`) so that it is no longer an integer but is instead a string. Converting numbers to strings for concatenation can be useful when dealing with zip codes or phone numbers. For example, you wouldn't want to perform addition between a country code and an area code, but you do want them to stay together.

When you combine two or more strings through concatenation, you are creating a new string that you can use throughout your program.

Storing Strings in Variables

Variables are symbols that you can use to store data in a program. You can think of them as an empty box that you fill with some data or value. Strings are data, so you can use them to fill up a variable. Declaring strings as variables can make it easier to work with strings throughout your Go programs.

To store a string inside a variable, simply assign a variable to a string. In this case, declare `s` as your variable:

```
s := "Sammy likes declaring strings."
```

Note: If you're familiar with other programming languages, you may have written the variable as `sammy`. Go, however, favors shorter variable names. Choosing `s` for the variable name in this case would be considered more appropriate for the style in which Go is written.

Now that you have the variable `s` set to that particular string, you can print the variable like so:

```
fmt.Println(s)
```

You will then receive the following output:

Output

```
Sammy likes declaring strings.
```

By using variables to stand in for strings, you do not have to retype a string each time you want to use it, making it more simple for you to work with and manipulate strings within your programs.

Conclusion

This tutorial went over the basics of working with the string data type in the Go programming language. Creating and printing strings, concatenating and replicating strings,

and storing strings in variables will provide you with the fundamentals to use strings in your Go programs.

How To Format Strings in Go

Written by Gopher Guides

As strings are often made up of written text, there are many instances when we may want to have greater control over how strings look to make them more readable for humans through punctuation, line breaks, and indentation.

In this tutorial, we'll go over some of the ways we can work with Go strings to make sure that all output text is formatted correctly.

String Literals

Let's first differentiate between a string literal and a string value. A string literal is what we see in the source code of a computer program, including the quotation marks. A string value is what we see when we call the `fmt.Println` function and run the program.

In the "Hello, World!" program, the string literal is `"Hello, World!"` while the string value is `Hello, World!` without the quotation marks. The string value is what we see as the output in a terminal window when we run a Go program.

But some string values may need to include quotation marks, like when we are quoting a source. Because string literals and string values are not equivalent, it is often necessary to add additional formatting to string literals to ensure that string values are displayed the way in which we intend.

Quotes

Because we can use back quotes (`` ` ``) or double quotes (`"`) within Go, it is simple to embed quotes within a string by using double quotes within a string enclosed by back quotes:

```
`Sammy says, "Hello!"`
```

Or, to use a back quote, you can enclose the string in double quotes:

```
"Sammy likes the `fmt` package for formatting strings.."
```

In the way we combine back quotes and double quotes, we can control the display of quotation marks and back quotes within our strings.

It's important to remember that using back quotes in Go creates a `raw` string literal, and using double quotes creates an `interpreted` string literal. To learn more about the difference, read the An Introduction to Working with Strings in Go tutorial.

Escape Characters

Another way to format strings is to use an escape character. Escape characters are used to tell the code that the following character has a special meaning. Escape characters all start with the backslash key (\) combined with another character within a string to format the given string a certain way.

Here is a list of several of the common escape characters:

Escape Character	How it formats
\\	Backslash
\"	Double Quote
	Line Break
Tab (horizontal indentation)	

Let's use an escape character to add the quotation marks to the example on quotation marks above, but this time we'll use double quotes to denote the string:

```
fmt.Println("Sammy says, \"Hello!\"")
```

Output

```
Sammy says, "Hello!"
```

By using the escape character `\"` we are able to use double quotes to enclose a string that includes text quoted between double quotes.

We can use the `\n` escape character to break lines without hitting the enter or return key:

```
fmt.Println("This string\nspans multiple\nlines.")
```

Output

```
This string
spans multiple
lines.
```

We can combine escape characters, too. Let's print a multi-line string and include tab spacing for an itemized list, for example:

```
fmt.Println("1.\tShark\n2.\tShrimp\n10.\tSquid")
```

Output

```
1.      Shark
2.      Shrimp
10.     Squid
```

The horizontal indentation provided with the `\t` escape character ensures alignment within the second column in the preceding example, making the output extremely readable for humans.

Escape characters are used to add additional formatting to strings that may be difficult or impossible to achieve. Without escape characters, you would not be able to construct the string `Sammy says, "I like to use the `fmt` package"`.

Multiple Lines

Printing strings on multiple lines can make text more readable to humans. With multiple lines, strings can be grouped into clean and orderly text, formatted as a letter, or used to maintain the linebreaks of a poem or song lyrics.

To create strings that span multiple lines, back quotes are used to enclose the string. Keep in mind that while this will preserve the line returns, it is also creating a `raw` string literal.

```
`
This string is on
multiple lines
within three single
quotes on either side.
`
```

You will notice if you print this that there is a leading and trailing return:

Output

```

This string is on
multiple lines
within three single
quotes on either side.

```

To avoid this, you need to put the first line immediately following the back quote and end the last with the back quote.

```
`This string is on
multiple lines
within three single
quotes on either side.`
```

If you need to create an interpreted string literal, this can be done with double quotes and the + operator, but you will need to insert your own line breaks.

```
"This string is on\n" +
"multiple lines\n" +
"within three single\n" +
"quotes on either side."
```

While back quotes can make it easier to print and read lengthy text, if you need an interpreted string literal, you will need to use double quotes.

Raw String Literals

What if we don't want special formatting within our strings? For example, we may need to compare or evaluate strings of computer code that use the backslash on purpose, so we won't want Go to use it as an escape character.

A raw string literal tells Go to ignore all formatting within a string, including escape characters.

We create a raw string by using back quotes around the string:

```
fmt.Println(`Sammy says,\"The balloon\'s color is red.\"`)
```

Output

```
Sammy says,\"The balloon\'s color is red.\"
```

By constructing a raw string by using back quotes around a given string, we can retain backslashes and other characters that are used as escape characters.

Conclusion

This tutorial went over several ways to format text in Go through working with strings. By using techniques such as escape characters or raw strings, we are able to ensure that the strings of our program are rendered correctly on-screen so that the end user is able to easily read all of the output text.

An Introduction to the Strings Package in Go

Written by Gopher Guides

Go's `string` package has several functions available to work with the string data type. These functions let us easily modify and manipulate strings. We can think of functions as being actions that we perform on elements of our code. Built-in functions are those that are defined in the Go programming language and are readily available for us to use.

In this tutorial, we'll review several different functions that we can use to work with strings in Go.

Making Strings Uppercase and Lowercase

The functions `strings.ToUpper` and `strings.ToLower` will return a string with all the letters of an original string converted to uppercase or lowercase letters. Because strings are immutable data types, the returned string will be a new string. Any characters in the string that are not letters will not be changed.

To convert the string `"Sammy Shark"` to be all uppercase, you would use the `strings.ToUpper` function:

```
ss := "Sammy Shark"
fmt.Println(strings.ToUpper(ss))
```

Output

```
SAMMY SHARK
```

To convert to lowercase:

```
fmt.Println(strings.ToLower(ss))
```

Output

```
sammy shark
```

Since you are using the `strings` package, you first need to import it into a program. To convert the string to uppercase and lowercase the entire program would be as follows:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
```

```
    ss := "Sammy Shark"
    fmt.Println(strings.ToUpper(ss))
    fmt.Println(strings.ToLower(ss))
}
```

The `strings.ToUpper` and `strings.ToLower` functions make it easier to evaluate and compare strings by making case consistent throughout. For example, if a user writes their name all lowercase, we can still determine whether their name is in our database by checking it against an all uppercase version.

String Search Functions

The `strings` package has a number of functions that help determine if a string contains a specific sequence of characters.

Function	Use
`strings.HasPrefix`	Searches the string from the beginning
`strings.HasSuffix`	Searches the string from the end
`strings.Contains`	Searches anywhere in the string
`strings.Count`	Counts how many times the string appears

The `strings.HasPrefix` and `strings.HasSuffix` allow you to check to see if a string starts or ends with a specific set of characters.

For example, to check to see if the string `"Sammy Shark"` starts with `Sammy` and ends with `Shark`:

```
ss := "Sammy Shark"
fmt.Println(strings.HasPrefix(ss, "Sammy"))
fmt.Println(strings.HasSuffix(ss, "Shark"))
```

Output

```
true
true
```

You would use the `strings.Contains` function to check if `"Sammy Shark"` contains the sequence `Sh`:

```
fmt.Println(strings.Contains(ss, "Sh"))
```

Output

```
true
```

Finally, to see how many times the letter `S` appears in the phrase `Sammy Shark`:

```
fmt.Println(strings.Count(ss, "S"))
```

Output

```
2
```

Note: All strings in Go are case sensitive. This means that `Sammy` is not the same as `sammy`.

Using a lowercase `s` to get a count from `Sammy Shark` is not the same as using uppercase `S`:

```
fmt.Println(strings.Count(ss, "s"))
```

Output

```
0
```

Because `S` is different than `s`, the count returned will be `0`.

String functions are useful when you want to compare or search strings in your program.

Determining String Length

The built-in function `len()` returns the number of characters in a string. This function is useful for when you need to enforce minimum or maximum password lengths, or to truncate larger strings to be within certain limits for use as abbreviations.

To demonstrate this function, we'll find the length of a sentence-long string:

```
import (
    "fmt"
    "strings"
)

func main() {
        openSource := "Sammy contributes to open source."
        fmt.Println(len(openSource))
}
```

Output

```
33
```

We set the variable `openSource` equal to the string `"Sammy contributes to open source."` and then passed that variable to the `len()` function with `len(openSource)`. Finally we passed the function into the `fmt.Println()` function so that we could see the program's output on the screen..

Keep in mind that the `len()` function will count any character bound by double quotation marks—including letters, numbers, whitespace characters, and symbols.

Functions for String Manipulation

The `strings.Join`, `strings.Split`, and `strings.ReplaceAll` functions are a few additional ways to manipulate strings in Go.

The `strings.Join` function is useful for combining a slice of strings into a new single string.

To create a comma-separated string from a slice of strings, we would use this function as per the following:

```
fmt.Println(strings.Join([]string{"sharks", "crustaceans", "plankton"}, ","))
```

Output

```
sharks,crustaceans,plankton
```

If we want to add a comma and a space between string values in our new string, we can simply rewrite our expression with a whitespace after the comma: `strings.Join([]string{"sharks", "crustaceans", "plankton"}, ", ")`.

Just as we can join strings together, we can also split strings up. To do this, we can use the `strings.Split` function and split on the spaces:

```
balloon := "Sammy has a balloon."
s := strings.Split(balloon, " ")
fmt.Println(s)
```

Output

```
[Sammy has a balloon]
```

The output is a slice of strings. Since `strings.Println` was used, it is hard to tell what the output is by looking at it. To see that it is indeed a slice of strings, use the `fmt.Printf` function with the `%q` verb to quote the strings:

```
fmt.Printf("%q", s)
```

Output

```
["Sammy" "has" "a" "balloon."]
```

Another useful function in addition to `strings.Split` is `strings.Fields`. The difference is that `strings.Fields` will ignore all whitespace, and will only split out the actual `fields` in a string:

```
data := "  username password     email  date"
fields := strings.Fields(data)
```

```go
fmt.Printf("%q", fields)
```

Output

```
["username" "password" "email" "date"]
```

The `strings.ReplaceAll` function can take an original string and return an updated string with some replacement.

Let's say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we would change the substring `"has"` from the original string `balloon` to `"had"` in a new string:

```go
fmt.Println(strings.ReplaceAll(balloon, "has", "had"))
```

Within the parentheses, first is `balloon` the variable that stores the original string; the second substring `"has"` is what we would want to replace, and the third substring `"had"` is what we would replace that second substring with. Our output would look like this when we incorporate this into a program:

Output

```
Sammy had a balloon.
```

Using the string function `strings.Join`, `strings.Split`, and `strings.ReplaceAll` will provide you with greater control to manipulate strings in Go.

Conclusion

This tutorial went through some of the common string package functions for the string data type that you can use to work with and manipulate strings in your Go programs.

You can learn more about other data types in Understanding Data Types and read more about strings in An Introduction to Working with Strings.

How To Use Variables and Constants in Go

Written by Gopher Guides

Variables are an important programming concept to master. They are symbols that stand in for a value you're using in a program.

This tutorial will cover some variable basics and best practices for using them within the Go programs you create.

Understanding Variables

In technical terms, a variable is assigning a storage location to a value that is tied to a symbolic name or identifier. We use the variable name to reference that stored value within a computer program.

We can think of a variable as a label that has a name on it, which you tie onto a value.



Variables in Go

Let's say we have an integer, `1032049348`, and we want to store it in a variable rather than continuously retype the long number over and over again. To achieve this, we can use a name that's easy to remember, like the variable `i`. To store a value in a variable, we use the following syntax:

```
i := 1032049348
```

We can think of this variable like a label that is tied to the value.



Go Variable Example

The label has the variable name `i` written on it, and is tied to the integer value `1032049348`.

The phrase `i := 1032049348` is a declaration and assignment statement that consists of a few parts:

- the variable name (`i`)
- the short variable declaration assignment (`:=`)
- the value that is being tied to the variable name (`1032049348`)
- the data type inferred by Go (`int`)

We'll see later how to explicitly set the type in the next section.

Together, these parts make up the statement that sets the variable `i` equal to the value of the integer `1032049348`.

As soon as we set a variable equal to a value, we initialize or create that variable. Once we have done that, we are ready to use the variable instead of the value.

Once we've set `i` equal to the value of `1032049348`, we can use `i` in the place of the integer, so let's print it out:

```
package main

import "fmt"

func main() {
    i := 1032049348
```

```
    fmt.Println(i)
}
```

Output

```
1032049348
```

We can also quickly and easily do math by using variables. With `i := 1032049348`, we can subtract the integer value `813` with the following syntax:

```
fmt.Println(i - 813)
```

Output

```
1032048535
```

In this example, Go does the math for us, subtracting 813 from the variable `i` to return the sum `1032048535`.

Speaking of math, variables can be set equal to the result of a math equation. You can also add two numbers together and store the value of the sum into the variable `x`:

```
x := 76 + 145
```

You may have noticed that this example looks similar to algebra. In the same way that we use letters and other symbols to represent numbers and quantities within formulas and equations, variables are symbolic names that represent the value of a data type. For correct Go syntax, you'll need to make sure that your variable is on the left side of any equations.

Let's go ahead and print `x`:

```
package main

import "fmt"

func main() {
    x := 76 + 145
    fmt.Println(x)
}
```

Output

```
221
```

Go returned the value 221 because the variable x was set equal to the sum of 76 and 145.

Variables can represent any data type, not just integers:

```
s := "Hello, World!"
f := 45.06
b := 5 > 9 // A Boolean value will return either true or false
array := [4]string{"item_1", "item_2", "item_3", "item_4"}
slice := []string{"one", "two", "three"}
m := map[string]string{"letter": "g", "number": "seven", "symbol": "&"}
```

If you print any of these variables, Go will return what that variable is equivalent to. Let's work with the assignment statement for the string slice data type:

```
package main

import "fmt"

func main() {
    slice := []string{"one", "two", "three"}
    fmt.Println(slice)
}
```

Output

```
[one two three]
```

We assigned the slice value of `[]string{"one", "two", "three"}` to the variable `slice`, and then used the `fmt.Println` function to print out that value by calling `slice`.

Variables work by carving out a little area of memory within your computer that accepts specified values that are then associated with that space.

Declaring Variables

In Go, there are several ways to declare a variable, and in some cases, more than one way to declare the exact same variable and value.

We can declare a variable called `i` of data type `int` without initialization. This means we will declare a space to put a value, but not give it an initial value:

```
var i int
```

This creates a variable declared as `i` of data type `int`.

We can initialize the value by using the equal (=) operator, like in the following example:

```go
var i int = 1
```

In Go, both of these forms of declaration are called long variable declarations.

We can also use short variable declaration:

```go
i := 1
```

In this case, we have a variable called `i`, and a data type of `int`. When we don't specify a data type, Go will infer the data type.

With the three ways to declare variables, the Go community has adopted the following idioms:

- Only use long form, `var i int`, when you're not initializing the variable.
- Use short form, `i := 1`, when declaring and initializing.
- If you did not desire Go to infer your data type, but you still want to use short variable declaration, you can wrap your value in your desired type, with the following syntax:

```go
i := int64(1)
```

It's not considered idiomatic in Go to use the long variable declaration form when we're initializing the value:

```go
var i int = 1
```

It's good practice to follow how the Go community typically declares variables so that others can seamlessly read your programs.

Zero Values

All built-in types have a zero value. Any allocated variable is usable even if it never has a value assigned. We can see the zero values for the following types:

```go
package main

import "fmt"

func main() {
    var a int
    var b string
    var c float64
    var d bool
```

```
    fmt.Printf("var a %T = %+v\n", a, a)
    fmt.Printf("var b %T = %q\n", b, b)
    fmt.Printf("var c %T = %+v\n", c, c)
    fmt.Printf("var d %T = %+v\n\n", d, d)
}
```

Output

```
var a int =  0
var b string = ""
var c float64 = 0
var d bool = false
```

We used the `%T` verb in the `fmt.Printf` statement. This tells the function to print the `data type` for the variable.

In Go, because all values have a `zero` value, we can't have `undefined` values like some other languages. For instance, a [`boolean`] in some languages could be `undefined`, `true`, or `false`, which allows for `three` states to the variable. In Go, we can't have more than `two` states for a boolean value.

Naming Variables: Rules and Style

The naming of variables is quite flexible, but there are some rules to keep in mind:

- Variable names must only be one word (as in no spaces).
- Variable names must be made up of only letters, numbers, and underscores (_).
- Variable names cannot begin with a number.

Following these rules, let's look at both valid and invalid variable names:

Valid	Invalid	Why Invalid
`userName`	`user-name`	Hyphens are not permitted
`name4`	`4name`	Cannot begin with a number
`user`	`$user`	Cannot use symbols
`userName`	`user name`	Cannot be more than one word

Furthermore, keep in mind when naming variables that they are case sensitive. These names `userName`, `USERNAME`, `UserName`, and `uSERnAME` are all completely different variables. It's best practice to avoid using similar variable names within a

program to ensure that both you and your collaborators—current and future—can keep your variables straight.

While variables are case sensitive, the case of the first letter of a variable has special meaning in Go. If a variable starts with an uppercase letter, then that variable is accessible outside the package it was declared in (or `exported`). If a variable starts with a lowercase letter, then it is only available within the package it is declared in.

```
var Email string
var password string
```

`Email` starts with an uppercase letter and can be accessed by other packages. `password` starts with a lowercase letter, and is only accessible inside the package it is declared in.

It is common in Go to use very terse (or short) variable names. Given the choice between using `userName` and `user` for a variable, it would be idiomatic to choose `user`.

Scope also plays a role in the terseness of the variable name. The rule is that the smaller the scope the variable exists in, the smaller the variable name:

```
names := []string{"Mary", "John", "Bob", "Anna"}
for i, n := range names {
    fmt.Printf("index: %d = %q\n", i, n)
}
```

We use the variable `names` in a larger scope, so it would be common to give it a more meaningful name to help remember what it means in the program. However, we use the `i` and `n` variables immediately in the next line of code, and then do not use them again.. Because of this, it won't confuse someone reading the code about where the variables are used, or what they mean.

Next, let's cover some notes about variable style. The style is to use `MixedCaps` or `mixedCaps` rather than underscores for multi-word names.

Conventional Style	Unconventional Style	Why Unconventional
`userName`	`user_name`	Underscores are not conventional
`i`	`index`	prefer `i` over `index` as it is shorter
`serveHTTP`	`serveHttp`	acronyms should be capitalized

The most important thing about style is to be consistent, and that the team you work on agrees to the style.

Reassigning Variables

As the word “variable” implies, we can change Go variables readily. This means that we can connect a different value with a previously assigned variable through reassignment. Being able to reassign is useful because throughout the course of a program we may need to accept user-generated values into already initialized variables. We may also need to change the assignment to something previously defined.

Knowing that we can readily reassign a variable can be useful when working on a large program that someone else wrote, and it isn’t clear what variables are already defined.

Let’s assign the value of 76 to a variable called `i` of type `int`, then assign it a new value of 42:

```go
package main

import "fmt"

func main() {
    i := 76
    fmt.Println(i)

    i = 42
    fmt.Println(i)
}
```

Output

```
76
42
```

This example shows that we can first assign the variable `i` with the value of an integer, and then reassign the variable `i` assigning it this time with the value of 42.

Note: When you declare and initialize a variable, you can use `:=`, however, when you want to simply change the value of an already declared variable, you only need to use the equal operator (`=`).

Because Go is a `typed` language, we can’t assign one type to another. For instance, we can’t assign the value `"Sammy"` to a variable of type `int`:

```
i := 72
i = "Sammy"
```

Trying to assign different types to each other will result in a compile-time error:

Output

```
cannot use "Sammy" (type string) as type int in assignment
```

Go will not allow us to use a variable name more than once:

```
var s string
var s string
```

Output

```
s redeclared in this block
```

If we try to use short variable declaration more than once for the same variable name we'll also receive a compilation error. This can happen by mistake, so understanding what the error message means is helpful:

```
i := 5
i := 10
```

Output

```
no new variables on left side of :=
```

Similarly to variable declaration, giving consideration to the naming of your variables will improve the readability of your program for you, and others, when you revisit it in the future.

Multiple Assignment

Go also allows us to assign several values to several variables within the same line. Each of these values can be of a different data type:

```
j, k, l := "shark", 2.05, 15
fmt.Println(j)
fmt.Println(k)
fmt.Println(l)
```

Output

```
shark
2.05
15
```

In this example, the variable `j` was assigned to the string `"shark"`, the variable `k` was assigned to the float `2.05`, and the variable `l` was assigned to the integer `15`.

This approach to assigning multiple variables to multiple values in one line can keep the number of lines in your code down. However, it's important to not compromise readability for fewer lines of code.

Global and Local Variables

When using variables within a program, it is important to keep variable scope in mind. A variable's scope refers to the particular places it is accessible from within the code of a given program. This is to say that not all variables are accessible from all parts of a given program—some variables will be global and some will be local.

Global variables exist outside of functions. Local variables exist within functions.

Let's take a look at global and local variables in action:

```go
package main

import "fmt"


var g = "global"

func printLocal() {
    l := "local"
    fmt.Println(l)
}

func main() {
    printLocal()
    fmt.Println(g)
}
```

Output

```
local
global
```

Here we use `var g = "global"` to create a global variable outside of the function. Then we define the function `printLocal()`. Inside of the function a local variable called `l` is assigned and then printed out. The program ends by calling `printLocal()` and then printing the global variable `g`.

Because `g` is a global variable, we can refer to it in `printLocal()`. Let's modify the previous program to do that:

```go
package main

import "fmt"


var g = "global"

func printLocal() {
    l := "local"
    fmt.Println(l)
    fmt.Println(g)
}

func main() {
    printLocal()
    fmt.Println(g)
}
```

Output

```
local
global
global
```

We start by declaring a global variable `g`, `var g = "global"`. In the `main` function, we call the function `printLocal`, which declares a local variable `l` and prints it out, `fmt.Println(l)`. Then, `printLocal` prints out the global variable `g`, `fmt.Println(g)`. Even though `g` wasn't defined within `printLocal`, it could still be

accessed because it was declared in a global scope. Finally, the `main` function prints out `g` as well.

Now let's try to call the local variable outside of the function:

```
package main

import "fmt"

var g = "global"

func printLocal() {
    l := "local"
    fmt.Println(l)
}

func main() {
    fmt.Println(l)
}
```

Output

```
undefined: l
```

We can't use a local variable outside of the function it is assigned in. If you try to do so, you'll receive a `undefined` error when you compile.

Let's look at another example where we use the same variable name for a global variable and a local variable:

```
package main

import "fmt"

var num1 = 5

func printNumbers() {
    num1 := 10
    num2 := 7

    fmt.Println(num1)
    fmt.Println(num2)
```

```
}

func main() {
    printNumbers()
    fmt.Println(num1)
}
```

Output

```
10
7
5
```

In this program, we declared the `num1` variable twice. First, we declared `num1` at the global scope, `var num1 = 5`, and again within the local scope of the `printNumbers` function, `num1 := 10`. When we print `num1` from the `main` program, we see the value of `5` printed out. This is because `main` only sees the global variable declaration. However, when we print out `num1` from the `printNumbers` function, it sees the local declaration, and will print out the value of `10`. Even though `printNumbers` creates a new variable called `num1` and assigned it a value of `10`, it does not affect the global instance of `num1` with the value of `5`.

When working with variables, you also need to consider what parts of your program will need access to each variables; adopting a global or local variable accordingly. Across Go programs, you'll find that local variables are typically more common.

Constants

Constants are like variables, except they can't be modified once they have been declared. Constants are useful for defining a value that will be used more than once in your program, but shouldn't be able to change.

For instance, if we wanted to declare the tax rate for a shopping cart system, we could use a constant and then calculate tax in different areas of our program. At some point in the future, if the tax rate changes, we only have to change that value in one spot in our program. If we used a variable, it is possible that we might accidentally change the value somewhere in our program, which would result in an improper calculation.

To declare a constant, we can use the following syntax:

```
const shark = "Sammy"
fmt.Println(shark)
```

Output

```
Sammy
```

If we try to modify a constant after it was declared, we'll get a compile-time error:

Output

```
cannot assign to shark
```

Constants can be `untyped`. This can be useful when working with numbers such as integer-type data. If the constant is `untyped`, it is explicitly converted, where `typed` constants are not. Let's see how we can use constants:

```
package main

import "fmt"

const (
    year     = 365
    leapYear = int32(366)
)

func main() {
    hours := 24
    minutes := int32(60)
    fmt.Println(hours * year)
    fmt.Println(minutes * year)
    fmt.Println(minutes * leapYear)
}
```

Output

```
8760
21900
21960
```

If you declare a constant with a type, it will be that exact type. Here when we declare the constant `leapYear`, we define it as data type `int32`. Therefore it is a `typed` constant, which means it can only operate with `int32` data types. The `year` constant we declare with no type, so it is considered `untyped`. Because of this, you can use it with any integer data type.

When `hours` was defined, it inferred that it was of type `int` because we did not explicitly give it a type, `hours := 24`. When we declared `minutes`, we explicitly declared it as an `int32`, `minutes := int32(60)`.

Now let's walk through each calculation and why it works:

```
hours * year
```

In this case, `hours` is an `int`, and `years` is untyped. When the program compiles, it explicitly converts `years` to an `int`, which allows the multiplication operation to succeed.

```
minutes * year
```

In this case, `minutes` is an `int32`, and `year` is untyped. When the program compiles, it explicitly converts `years` to an `int32`, which allows the multiplication operation to succeed.

```
minutes * leapYear
```

In this case, `minutes` is an `int32`, and `leapYear` is a typed constant of `int32`. There is nothing for the compiler to do this time as both variables are already of the same type.

If we try to multiply two types that are `typed` and not compatible, the program will not compile:

```
fmt.Println(hours * leapYear)
```

Output

```
invalid operation: hours * leapYear (mismatched types int and int32)
```

In this case, `hours` was inferred as an `int`, and `leapYear` was explicitly declared as an `int32`. Because Go is a typed language, an `int` and an `int32` are not compatible for mathematical operations. To multiply them, you would need [to convert one to a `int32` or an `int`].

Conclusion

In this tutorial we reviewed some of the common use cases of variables within Go. Variables are an important building block of programming, serving as symbols that stand in for the value of a data type we use in a program.

How To Convert Data Types in Go

Written by Gopher Guides

In Go, data types are used to classify one particular type of data, determining the values that you can assign to the type and the operations you can perform on it. When programming, there are times when you will need to convert values between types in order to manipulate values in a different way. For example, you may need to concatenate numeric values with strings, or represent decimal places in numbers that were initialized as integer values. User-generated data is often automatically assigned the string data type, even if it consists of numbers; in order to perform mathematical operations in this input, you would have to convert the string to a numeric data type.

Since Go is a statically typed language, data types are bound to variables rather than values. This means that, if you define a variable as an `int`, it can only be an `int`; you can't assign a `string` to it without converting the data type of the variable. The static nature of data types in Go places even more importance on learning the ways to convert them.

This tutorial will guide you through converting numbers and strings, as well as provide examples to help familiarize yourself with different use cases.

Converting Number Types

Go has several numeric types to choose from. Primarily they break out into two general types: integers and floating-point numbers.

There are many situations in which you may want to convert between numeric types. Converting between different sizes of numeric types can help optimize performance for specific kinds of system architecture. If you have an integer from another part of your code and want to do division on it, you may want to convert the integer to a float to preserve the precision of the operation. Additionally, working with time durations usually involves integer conversion. To address these situations, Go has built-in type conversions for most numeric types.

Converting Between Integer Types

Go has many integer data types to pick from. When to use one over the other is typically more about performance; however, there will be times when you will need to convert from one integer type to another. For example, Go sometimes automatically generates numeric values as `int`, which may not match your input value. If your input value were `int64`, you would not be able to use the `int` and the `int64` numbers in the same mathematical expression until you converted their data types to match.

Assume that you have an `int8` and you need to convert it to an `int32`. You can do this by wrapping it in the `int32()` type conversion:

```
var index int8 = 15

var bigIndex int32

bigIndex = int32(index)

fmt.Println(bigIndex)
```

Output

```
15
```

This code block defines `index` as an `int8` data type and `bigIndex` as an `int32` data type. To store the value of `index` in `bigIndex`, it converts the data type to an `int32`. This is done by wrapping the `int32()` conversion around the `index` variable.

To verify your data types, you could use the `fmt.Printf` statement and the `%T` verb with the following syntax:

```
fmt.Printf("index data type:    %T\n", index)
fmt.Printf("bigIndex data type: %T\n", bigIndex)
```

Output

```
index data type:    int8
bigIndex data type: int32
```

Since this uses the `%T` verb, the print statement outputs the type for the variable, and not the actual value of the variable. This way, you can confirm the converted data type.

You can also convert from a larger bit-size integer to a smaller bit-size integer:

```
var big int64 = 64

var little int8

little = int8(big)

fmt.Println(little)
```

Output

```
64
```

Keep in mind that when converting integers you could potentially exceed the maximum value of the data type and wraparound:

```go
var big int64 = 129
var little = int8(big)
fmt.Println(little)
```

Output

```
-127
```

A wraparound happens when the value is converted to a data type that is too small to hold it. In the preceding example, the 8-bit data type `int8` did not have enough space to hold the 64-bit variable `big`. Care should always be taken when converting from a larger number data type to a smaller number data type so that you do not truncate the data by accident.

Converting Integers to Floats

Converting integers to floats in Go is similar to converting one integer type to another. You can use the built-in type conversions by wrapping `float64()` or `float32()` around the integer you are converting:

```go
var x int64 = 57

var y float64 = float64(x)

fmt.Printf("%.2f\n", y)
```

Output

```
57.00
```

This code declares a variable `x` of type `int64` and initializes its value to `57`.

```go
var x int64 = 57
```

Wrapping the `float64()` conversion around `x` will convert the value of `57` to a float value of `57.00`.

```go
var y float64 = float64(x)
```

The `%.2f` print verb tells `fmt.Printf` to format the float with two decimals.

You can also use this process on a variable. The following code declares `f` as equal to `57`, and then prints out the new float:

```go
var f float64 = 57
fmt.Printf("%.2f\n", f)
```

Output

```
57.00
```

By using either `float32()` or `float64()`, you can convert integers to floats. Next, you will learn how to convert floats to integers.

Converting Floats to Integers

Go can convert floats to integers, but the program will lose the precision of the float.

Wrapping floats in `int()`, or one of its architecture-independent data types, works similarly to when you used it to convert from one integer type to another. You can add a floating-point number inside of the parentheses to convert it to an integer:

```
var f float64 = 390.8
var i int = int(f)

fmt.Printf("f = %.2f\n", f)
fmt.Printf("i = %d\n", i)
```

Output

```
f = 390.80
i = 390
```

This syntax would convert the float `390.8` to the integer `390`, dropping the decimal place.

You can also use this with variables. The following code declares `b` as equal to `125.0` and `c` as equal to `390.8`, then prints them out as integers. Short variable declaration (`:=`) shortens up the syntax:

```
b := 125.0
c := 390.8

fmt.Println(int(b))
fmt.Println(int(c))
```

Output

```
125
390
```

When converting floats to integers with the `int()` type, Go cuts off the decimal and remaining numbers of a float to create an integer. Note that, even though you may want to round 390.8 up to 391, Go will not do this through the `int()` type. Instead, it will drop the decimal.

Numbers Converted Through Division

When dividing integer types in Go the result will also be an integer type, with the modulus, or remainder, dropped:

```
a := 5 / 2
fmt.Println(a)
```

Output

```
2
```

If, when dividing, any of the number types are a float, then all of the types will automatically be declared as a float:

```
    a := 5.0 / 2
    fmt.Println(a)
```

Output

```
2.5
```

This divides the float `5.0` by the integer `2`, and the answer `2.5` is a float that retains the decimal precision.

In this section, you have converted between different number data types, including differing sizes of integers and floating-point numbers. Next, you will learn how to convert between numbers and strings.

Converting with Strings

A string is a sequence of one or more characters (letters, numbers, or symbols). Strings are a common form of data in computer programs, and you may need to convert strings to numbers or numbers to strings fairly often, especially when you are taking in user-generated data.

Converting Numbers to Strings

You can convert numbers to strings by using the `strconv.Itoa` method from the `strconv` package in the Go standard libary. If you pass either a number or a variable into the parentheses of the method, that numeric value will be converted into a string value.

First, let's look at converting integers. To convert the integer `12` to a string value, you can pass `12` into the `strconv.Itoa` method:

```
package main

import (
    "fmt"
    "strconv"
)
```

```
func main() {
    a := strconv.Itoa(12)
    fmt.Printf("%q\n", a)
}
```

When running this program, you'll receive the following output:

Output

```
"12"
```

The quotes around the number 12 signify that the number is no longer an integer but is now a string value.

You used the `:=` assignment operator to both declare a new variable with the name of `a` and assign the value returned from the `strconv.Itoa()` function. In this case, you assigned the value `12` to your variable. You also used the `%q` verb in the `fmt.Printf` function, which tells the function to quote the string provided.

With variables you can begin to see how practical it can be to convert integers to strings. Say you want to keep track of a user's daily programming progress and are inputting how many lines of code they write at a time. You would like to show this feedback to the user and will be printing out string and integer values at the same time:

```
package main

import (
    "fmt"
)

func main() {
    user := "Sammy"
    lines := 50

    fmt.Println("Congratulations, " + user + "! You just wrote " + lines + " 
}
```

When you run this code, you'll receive the following error:

Output

```
invalid operation: ("Congratulations, " + user + "! You just wrote ") +
lines (mismatched types string and int)
```

You're not able to concatenate strings and integers in Go, so you'll have to convert the variable `lines` to be a string value:

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    user := "Sammy"
    lines := 50

    fmt.Println("Congratulations, " + user + "! You just wrote " +
strconv.Itoa(lines) + " lines of code.")
}
```

Now, when you run the code, you'll receive the following output that congratulates your user on their progress:

Output

```
Congratulations, Sammy! You just wrote 50 lines of code.
```

If you are looking to convert a float to a string rather than an integer to a string, you follow similar steps and format. When you pass a float into the `fmt.Sprint` method, from the `fmt` package in the Go standard library, a string value of the float will be returned. You can use either the float value itself or a variable:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(fmt.Sprint(421.034))

    f := 5524.53
    fmt.Println(fmt.Sprint(f))
}
```

Output

```
421.034
5524.53
```

You can test to make sure it's right by concatenating with a string:

```go
package main

import (
    "fmt"
)

func main() {
    f := 5524.53
    fmt.Println("Sammy has " + fmt.Sprint(f) + " points.")
}
```

Output

```
Sammy has 5524.53 points.
```

You can be sure your float was properly converted to a string because the concatenation was performed without error.

Converting Strings to Numbers

Strings can be converted to numbers by using the `strconv` package in the Go standard library. The `strconv` package has functions for converting both integer and float number types. This is a very common operation when accepting input from the user. For example, if you had a program that asked for a person's age, when they type the response in, it is captured as a `string`. You would then need to convert it to an `int` to do any math with it.

If your string does not have decimal places, you'll most likely want to convert it to an integer by using the `strconv.Atoi` function. If you know you will use the number as a float, you would use `strconv.ParseFloat`.

Let's use the example of the user Sammy keeping track of lines of code written each day. You may want to manipulate those values with math to provide more interesting feedback for the user, but those values are currently stored in strings:

```go
package main

import (
    "fmt"
)

func main() {
    lines_yesterday := "50"
    lines_today := "108"
```

```
    lines_more := lines_today - lines_yesterday

    fmt.Println(lines_more)
}
```

Output

```
invalid operation: lines_today - lines_yesterday (operator - not defined on
string)
```

Because the two numeric values were stored in strings, you received an error. The operand - for subtraction is not a valid operand for two string values.

Modify the code to include the `strconv.Atoi()` method that will convert the strings to integers, which will allow you to do math with values that were originally strings. Because there is a potential to fail when converting a string to an integer, you have to check for any errors. You can use an `if` statement to check if your conversion was successful.

```
package main

import (
    "fmt"
    "log"
    "strconv"
)

func main() {
    lines_yesterday := "50"
    lines_today := "108"

    yesterday, err := strconv.Atoi(lines_yesterday)
    if err != nil {
        log.Fatal(err)
    }

    today, err := strconv.Atoi(lines_today)
    if err != nil {
        log.Fatal(err)
    }
    lines_more := today - yesterday

    fmt.Println(lines_more)
}
```

Because it is possible for a string to not be a number, the `strconv.Atoi()` method will return both the converted type, as well as a potential error. When converting from `lines_yesterday` with the `strconv.Atoi` function, you have to check the `err` return value to ensure that the value was converted. If the `err` is not `nil`, it means that `strconv.Atoi` was unable to successfully convert the string value to an integer. In this example, you used an `if` statement to check for the error, and if an error was returned, you used `log.Fatal` to log the error and exit the program.

When you run the preceding code, you will get:

Output

```
58
```

Now try to convert a string that is not a number:

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    a := "not a number"
    b, err := strconv.Atoi(a)
    fmt.Println(b)
    fmt.Println(err)
}
```

You will get the following error:

Output

```
0
strconv.Atoi: parsing "not a number": invalid syntax
```

Because `b` was declared, but `strconv.Atoi` failed to make a conversion, a value was never assigned to `b`. Notice that `b` has the value of `0`. This is because Go has default values, referred to as zero values in Go. `strconv.Atoi` provides an error describing why it failed to convert the string as well.

Converting Strings and Bytes

Strings in Go are stored as a slice of bytes. In Go, you can convert between a slice of bytes and a string by wrapping it in the corresponding conversions of `[]byte()` and `string()`:

```go
package main

import (
	"fmt"
)

func main() {
	a := "my string"

	b := []byte(a)

	c := string(b)

	fmt.Println(a)

	fmt.Println(b)

	fmt.Println(c)
}
```

Here you have stored a string value in `a`, then converted it to a slice of bytes `b`, then converted the slice of bytes back to a string as `c`. You then print `a`, `b`, and `c` to the screen:

Output

```
my string
[109 121 32 115 116 114 105 110 103]
my string
```

The first line of output is the original string `my string`. The second line printed out is the byte slice that makes up the original string. The third line shows that the byte slice can be safely converted back into a string and printed back out.

Conclusion

This Go tutorial demonstrated how to convert several of the important native data types to other data types, primarily through built-in methods. Being able to convert data types in Go will allow you to do things like accept user input and do math across different number types. Later on, when you are using Go to write programs that accept data from many different sources like databases and APIs, you will use these conversion methods to ensure you can act on your data. You will also be able to optimize storage by converting data to smaller data types.

If you would like a deeper analysis of data types in Go, check out our Understanding Data Types in Go article.

How To Do Math in Go with Operators

Written by Gopher Guides

Numbers are common in programming. They are used to represent things such as: screen-size dimensions, geographic locations, money and points, the amount of time that passes in a video, positions of game avatars, colors through assigning numeric codes, and so on.

Effectively performing mathematical operations in programming is an important skill to develop because of how frequently you'll work with numbers. Though a high-level understanding of mathematics can certainly help you become a better programmer, it is not a prerequisite. If you don't have a background in mathematics, try to think of math as a tool to accomplish what you would like to achieve, and as a way to improve your logical thinking.

We'll be working with two of Go's most used numeric data types, integers and floats:

- Integers are whole numbers that can be positive, negative, or 0 (…, `-1`, `0`, `1`, …).
- Floats are real numbers that contain a decimal point, like `9.0` or `-2.25`..

This tutorial will review operators that we can use with number data types in Go.

Operators

An operator is a symbol or function that indicates an operation. For example, in math the plus sign or + is the operator that indicates addition.

In Go, we will see some familiar operators that are brought over from math. However, other operators we will use are specific to computer programming.

Here is a quick reference table of math-related operators in Go. We'll be covering all of the following operations in this tutorial.

Operation	What it returns
`x + y`	Sum of x and y
`x - y`	Difference of x and y
`-x`	Changed sign of x
`+x`	Identity of x
`x * y`	Product of x and y
`x / y`	Quotient of x and y
`x % y`	Remainder of x / y

We'll also be covering compound assignment operators, including += and *=, that combine an arithmetic operator

with the = operator.

Addition and Subtraction

In Go, addition and subtraction operators perform just as they do in mathematics. In fact, you can use the Go programming language as a calculator.

Let's look at some examples, starting with integers:

```
fmt.Println(1 + 5)
```

Output

```
6
```

Instead of passing integers directly into the `fmt.Println` statement, we can initialize variables to stand for integer values by using syntax like the following:

```
a := 88
b := 103

fmt.Println(a + b)
```

Output

```
191
```

Because integers can be both positive and negative numbers (and 0 too), we can add a negative number with a positive number:

```
c := -36
d := 25

fmt.Println(c + d)
```

Output

```
-11
```

Addition will behave similarly with floats:

```
e := 5.5
f := 2.5

fmt.Println(e + f)
```

Output

```
8
```

Because we added two floats together, Go returned a float value with a decimal place. However, since the decimal place is zero in this case, `fmt.Println` dropped the decimal formatting. To properly format the output, we can use `fmt.Printf` and the verb `%.2f`, which will format to two decimal places, like this example:

```
fmt.Printf("%.2f", e + f)
```

Output

```
8.00
```

The syntax for subtraction is the same as for addition, except we change our operator from the plus sign (+) to the minus sign (-):

```
g := 75.67
h := 32.0

fmt.Println(g - h)
```

Output

```
43.67
```

In Go, we can only use operators on the same data types. We can't add an `int` and a [`float64`]:

```
i := 7
j := 7.0
fmt.Println(i + j)
```

Output

```
i + j (mismatched types int and float64)
```

Trying to use operators on data types that are not the same will result in a compiler error.

Unary Arithmetic Operations

A unary mathematical expression consists of only one component or element. In Go we can use the plus and minus signs as a single element paired with a value to: return the value's identity (+), or change the sign of the value (-).

Though not commonly used, the plus sign indicates the identity of the value. We can use the plus sign with positive values:

```
i := 3.3
fmt.Println(+i)
```

Output

```
3.3
```

When we use the plus sign with a negative value, it will also return the identity of that value, and in this case it would be a negative value:

```
j := -19
fmt.Println(+j)
```

Output

```
-19
```

With a negative value the plus sign returns the same negative value.

The minus sign, however, changes the sign of a value. So, when we pass a positive value we'll find that the minus sign

before the value will return a negative value:

```
k := 3.3
fmt.Println(-k)
```

Output

```
-3.3
```

Alternatively, when we use the minus sign unary operator with a negative value, a positive value will be returned:

```
j := -19
fmt.Println(-j)
```

Output

```
19
```

The unary arithmetic operations indicated by the plus sign and minus sign will return either the value's identity in the case of `+i`, or the opposite sign of the value as in `-i`.

Multiplication and Division

Like addition and subtraction, multiplication and division will look very similar to how they do in mathematics. The sign we'll use in Go for multiplication is * and the sign we'll use for division is `/`.

Here's an example of doing multiplication in Go with two float values:

```
k := 100.2
l := 10.2

fmt.Println(k * l)
```

Output

```
1022.04
```

In Go, division has different characteristics depending on the numeric type we're dividing.

If we're dividing integers, Go's `/` operator performs floor division, where for the quotient x the number returned is the largest integer less than or equal to x.

If you run the following example of dividing `80 / 6`, you'll receive `13` as the output and the data type will be `int`:

```
package main

import (
    "fmt"
)

func main() {
    m := 80
    n := 6
```

```
    fmt.Println(m / n)
}
```

Output

```
13
```

If the desired output is a float, you have to explicitly convert the values before dividing.

You can do this by wrapping your desired float type of `float32()` or `float64()` around your values:

```
package main

import (
    "fmt"
)

func main() {
    s := 80
    t := 6
    r := float64(s) / float64(t)
    fmt.Println(r)
}
```

Output

```
13.333333333333334
```

Modulo

The `%` operator is the modulo, which returns the remainder rather than the quotient after division. This is useful for finding numbers that are multiples of the same number.

Let's look at an example of the modulo:

```
o := 85
p := 15

fmt.Println(o % p)
```

Output

```
10
```

To break this down, `85` divided by `15` returns the quotient of `5` with a remainder of `10`. Our program returns the value `10` here, because the modulo operator returns the remainder of a division expression.

To do modulus math with `float64` data types, you'll use the `Mod` function from the `math` package:

```
package main

import (
    "fmt"
    "math"
)
```

```
func main() {
    q := 36.0
    r := 8.0

    s := math.Mod(q, r)

    fmt.Println(s)
}
```

Output

```
4
```

Operator Precedence

In Go, as in mathematics, we need to keep in mind that operators will be evaluated in order of precedence, not from left to right or right to left.

If we look at the following mathematical expression:

```
u = 10 + 10 * 5
```

We may read it left to right, but multiplication will be done first, so if we were to print `u`, we would receive the following value:

Output

```
60
```

This is because `10 * 5` evaluates to `50`, and then we add `10` to return `60` as the final result.

If instead we would like to add the value `10` to `10`, then multiply that sum by `5`, we use parentheses in Go just like we would in math:

```
u := (10 + 10) * 5
fmt.Println(u)
```

Output

```
100
```

One way to remember the order of operation is through the acronym PEMDAS:

Order	Letter	Stands for
1	P	Parentheses
2	E	Exponent
3	M	Multiplication
4	D	Division
5	A	Addition
6	S	Subtraction

You may be familiar with another acronym for the order of operations, such as BEDMAS or BODMAS. Whatever acronym works best for you, try to keep it in mind when

performing math operations in Go so that the results that you expect are returned.

Assignment Operators

The most common assignment operator is one you have already used: the equals sign =. The = assignment operator assigns the value on the right to a variable on the left. For example, `v = 23` assigns the value of the integer 23 to the variable v.

When programming, it is common to use compound assignment operators that perform an operation on a variable's value and then assign the resulting new value to that variable. These compound operators combine an arithmetic operator with the = operator. Therefore, for addition we'll combine + with = to get the compound operator +=. Let's see what that looks like:

```
w := 5
w += 1
fmt.Println(w)
```

Output

```
6
```

First, we set the variable w equal to the value of 5, then we use the += compound assignment operator to add the

right number to the value of the left variable, and then assign the result to `w`.

Compound assignment operators are used frequently in the case of `for` loops, which you'll use when you want to repeat a process several times:

```
package main

import "fmt"

func main() {

    values := []int{0, 1, 2, 3, 4, 5, 6}

    for _, x := range values {

        w := x

        w *= 2

        fmt.Println(w)
    }

}
```

Output

```
0
2
4
6
8
10
12
```

By using a `for` loop to iterate over the slice called `values`, you were able to automate the process of the `*=` operator that multiplied the variable `w` by the number `2` and then assigned the result back into the variable `w`.

Go has a compound assignment operator for each of the arithmetic operators discussed in this tutorial.

To add then assign the value:

```
y += 1
```

To subtract then assign the value:

```
y -= 1
```

To multiply then assign then value:

```
y *= 2
```

To divide then assign the value:

```
y /= 3
```

To return the remainder then assign the value:

```
y %= 3
```

Compound assignment operators can be useful when things need to be incrementally increased or decreased, or

when you need to automate certain processes in your program.

Conclusion

This tutorial covered many of the operators you'll use with the integer and float numeric data types. You can learn more about different data types in Understanding Data Types in Go and How To Convert Data Types.

Understanding Boolean Logic in Go

Written by Gopher Guides

The Boolean data type (`bool`) can be one of two values, either true or false. Booleans are used in programming to make comparisons and to control the flow of the program.

Booleans represent the truth values that are associated with the logic branch of mathematics, which informs algorithms in computer science. Named for the mathematician George Boole, the word Boolean always begins with a capitalized `B`.

The data type in Go for Boolean is `bool`, all lowercase. The values `true` and `false` will always be with a lowercase `t` and `f` respectively, as they are special values in Go.

This tutorial will cover the basics you'll need to understand how the `bool` data type works, including Boolean comparison, logical operators, and truth tables.

Comparison Operators

In programming, comparison operators are used to compare values and evaluate down to a single Boolean value of either true or false.

The table below shows Boolean comparison operators.

Operator	What it means
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

To understand how these operators work, let's assign two integers to two variables in a Go program:

```
x := 5
y := 8
```

In this example, since `x` has the value of `5`, it is less than `y` which has the value of `8`.

Using those two variables and their associated values, let's go through the operators from the preceding table. In this program, you'll ask Go to print out whether each comparison operator evaluates to either true or false. To help better understand this output, you'll have Go also print a string to show you what it's evaluating:

```
package main
```

```
import "fmt"

func main() {
    x := 5
    y := 8

    fmt.Println("x == y:", x == y)
    fmt.Println("x != y:", x != y)
    fmt.Println("x < y:", x < y)
    fmt.Println("x > y:", x > y)
    fmt.Println("x <= y:", x <= y)
    fmt.Println("x >= y:", x >= y)
}
```

Output

```
x == y: false
x != y: true
x < y: true
x > y: false
x <= y: true
x >= y: false
```

Following mathematical logic, Go has evaluated the following from the expressions:

- Is 5 (x) equal to 8 (y)? false
- Is 5 not equal to 8? true
- Is 5 less than 8? true
- Is 5 greater than 8? false
- Is 5 less than or equal to 8? true
- Is 5 not less than or equal to 8? false

Although integers were used here, you could substitute them with float values.

Strings can also be used with Boolean operators. They are case-sensitive unless you use an additional string method.

You can look at how strings are compared in practice:

```
Sammy := "Sammy"
sammy := "sammy"

fmt.Println("Sammy == sammy: ", Sammy == sammy)
```

Output

```
Sammy == sammy:  false
```

The string Sammy is not equal to the string sammy, because they are not exactly the same; one starts with an uppercase S and the other with a lowercase s. But, if you add another variable that is assigned the value of Sammy, then they will evaluate to equal:

```
Sammy := "Sammy"
sammy := "sammy"
alsoSammy := "Sammy"

fmt.Println("Sammy == sammy: ", Sammy == sammy)
fmt.Println("Sammy == alsoSammy", Sammy == alsoSammy)
```

Output

```
Sammy == sammy:  false
Sammy == alsoSammy true
```

You can also use the other comparison operators including > and < to compare two strings. Go will compare these strings lexicographically using the ASCII values of the characters.

You can also evaluate Boolean values with comparison operators:

```
t := true
f := false

fmt.Println("t != f: ", t != f)
```

Output

```
t != f:  true
```

The preceding code block evaluated that true is not equal to false.

Note the difference between the two operators = and ==.

```
x = y   // Sets x equal to y
x == y  // Evaluates whether x is equal to y
```

The first = is the assignment operator, which will set one value equal to another. The second, ==, is a comparison operator and will evaluate whether two values are equal.

Logical Operators

There are two logical operators that are used to compare values. They evaluate expressions down to Boolean values, returning either true or false. These operators are &&, ||, and !, and are defined in the list below:

- && (`x && y`) is the and operator. It is true if both statements are true.
- || (`x || y`) is the `or` operator. It is true if at least one statement is true.
- `!` (`!x`) is the `not` operator. It is true only if the statement is false.

Logical operators are typically used to evaluate whether two or more expressions are true or not true. For example, they can be used to determine if the grade is passing and that the student is registered in the course, and if both cases are true, then the student will be assigned a grade in the system. Another example would be to determine whether a user is a valid active customer of an online shop based on whether they have store credit or have made a purchase in the past 6 months.

To understand how logical operators work, let's evaluate three expressions:

```
fmt.Println((9 > 7) && (2 < 4))   // Both original expressions are true
fmt.Println((8 == 8) || (6 != 6)) // One original expression is true
fmt.Println(!(3 <= 1))            // The original expression is false
```

Output

```
true
true
true
```

In the first case, `fmt.Println((9 > 7) && (2 < 4))`, both `9 > 7` and `2 < 4` needed to evaluate to true since the and operator was used.

In the second case, `fmt.Println((8 == 8) || (6 != 6))`, since `8 == 8` evaluated to true, it did not make a difference that `6 != 6` evaluates to false because the `or` operator was used. If you had used the and operator, this would evaluate to false.

In the third case, `fmt.Println(!(3 <= 1))`, the `not` operator negates the false value that `3 <=1` returns.

Let's substitute floats for integers and aim for false evaluations:

```
fmt.Println((-0.2 > 1.4) && (0.8 < 3.1))  // One original expression is false
fmt.Println((7.5 == 8.9) || (9.2 != 9.2)) // Both original expressions are fa
fmt.Println(!(-5.7 <= 0.3))               // The original expression is true
```

In this example:

- `and` must have at least one false expression evaluate to false.
- `or` must have both expressions evaluate to false.
- `!` must have its inner expression be true for the new expression to evaluate to false.

If these results seem unclear to you, go through some truth tables for further clarification.

You can also write compound statements using &&, `||`, and `!`:

```
!((-0.2 > 1.4) && ((0.8 < 3.1) || (0.1 == 0.1)))
```

Take a look at the inner-most expression first: `(0.8 < 3.1) || (0.1 == 0.1)`. This expression evaluates to `true` because both mathematical statements are `true`.

Next, Go takes the returned value `true` and combines it with the next inner expression: `(-0.2 > 1.4) && (true)`. This example returns `false` because the mathematical statement `-0.2 > 1.4` is false, and (`false`) and (`true`) returns `false`.

Finally, we have the outer expression: `!(false)`, which evaluates to `true`, so the final returned value if we print this statement out is:

Output

```
true
```

The logical operators &&, `||`, and `!` evaluate expressions and return Boolean values.

Truth Tables

There is a lot to learn about the logic branch of mathematics, but you can selectively learn some of it to improve your algorithmic thinking when programming.

The following are truth tables for the comparison operator `==`, and each of the logic operators &&, `||` and `!`. While you may be able to reason them out, it can also be helpful to memorize them as that can make your programming decision-making process quicker.

== (equal) Truth Table

x	==	y	Returns
true	==	true	true
true	==	false	false
false	==	true	false
false	==	false	true

&& (and) Truth Table

x	and	y	Returns
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false

|| (or) Truth Table

x	or	y	Returns
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

! (not) Truth Table

not	x	Returns
not	true	false
not	false	true

Truth tables are common mathematical tables used in logic, and are useful to keep in mind when constructing algorithms (instructions) in computer programming.

Using Boolean Operators for Flow Control

To control the stream and outcomes of a program in the form of flow control statements, you can use a condition followed by a clause.

A condition evaluates down to a Boolean value of true or false, presenting a point where a decision is made in the program. That is, a condition would tell you if something evaluates to true or false.

The clause is the block of code that follows the condition and dictates the outcome of the program. That is, it is the "do this" part of the construction "If `x` is `true`, then do this."

The code block below shows an example of comparison operators working in tandem with conditional statements to control the flow of a Go program:

```go
if grade >= 65 {                 // Condition
    fmt.Println("Passing grade") // Clause
} else {
    fmt.Println("Failing grade")
}
```

This program will evaluate whether each student's grade is passing or failing. In the case of a student with a grade of `83`, the first statement will evaluate to `true`, and the print statement of `Passing grade` will be triggered. In the case of a student with a grade of `59`, the first statement will evaluate to `false`, so the program will move on to execute the print statement tied to the else expression: `Failing grade`.

Boolean operators present conditions that can be used to decide the eventual outcome of a program through flow control statements.

Conclusion

This tutorial went through comparison and logical operators belonging to the Boolean type, as well as truth tables and using Booleans for program flow control.

Understanding Maps in Go

Written by Gopher Guides

Most modern programming languages have the concept of a dictionary or a hash type. These types are commonly used to store data in pairs with a key that maps to a value.

In Go, the map data type is what most programmers would think of as the dictionary type. It maps keys to values, making key-value pairs that are a useful way to store data in Go. A map is constructed by using the keyword `map` followed by the key data type in square brackets `[ ]`, followed by the value data type. The key-value pairs are then placed inside curly braces on either side { }:

```
map[key]value{}
```

You typically use maps in Go to hold related data, such as the information contained in an ID. A map with data looks like this:

```
map[string]string{"name": "Sammy", "animal": "shark", "color": "blue", "locat
```

In addition to the curly braces, there are also colons throughout the map that connect the key-value pairs. The words to the left of the colons are the keys. Keys can be any comparable type in Go, like `strings`, `ints`, and so on.

The keys in the example map are:

- `"name"`
- `"animal"`
- `"color"`
- `"location"`

The words to the right of the colons are the values. Values can be any data type. The values in the example map are:

- `"Sammy"`
- `"shark"`
- `"blue"`
- `"ocean"`

Like the other data types, you can store the map inside a variable, and print it out:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark", "color": "blue
fmt.Println(sammy)
```

This would be your output:

Output

```
map[animal:shark color:blue location:ocean name:Sammy]
```

The order of the key-value pairs may have shifted. In Go, the map data type is unordered. Regardless of the order, the key-value pairs will remain intact, enabling you to access data based on their relational meaning.

Accessing Map Items

You can call the values of a map by referencing the related keys. Since maps offer key-value pairs for storing data, they can be important and useful items in your Go program.

If you want to isolate Sammy's username, you can do so by calling `sammy["name"]`; the variable holding your map and the related key. Let's print that out:

```
fmt.Println(sammy["name"])
```

And receive the value as output:

Output

```
Sammy
```

Maps behave like a database; instead of calling an integer to get a particular index value as you would with a slice, you assign a value to a key and call that key to get its related value.

By invoking the key `"name"` you receive the value of that key, which is `"Sammy"`.

Similarly you can call the remaining values in the `sammy` map using the same format:

```
fmt.Println(sammy["animal"])
// returns shark

fmt.Println(sammy["color"])
// returns blue

fmt.Println(sammy["location"])
// returns ocean
```

By making use of the key-value pairs in map data types, you can reference keys to retrieve values.

Keys and Values

Unlike some programming languages, Go does not have any convenience functions to list out the keys or values of a map. An example of this would be Python's `.keys()` method for dictionaries. It does, however, allow for iteration by using the `range` operator:

```
for key, value := range sammy {
    fmt.Printf("%q is the key for the value %q\n", key, value)
}
```

When ranging through a map in Go, it'll return two values. The first value will be the key, and the second value will be the value. Go will create these variables with the correct

data type. In this case, the map key was a `string` so `key` will also be a string. The `value` is also a string:

Output

```
"animal" is the key for the value "shark"
"color" is the key for the value "blue"
"location" is the key for the value "ocean"
"name" is the key for the value "Sammy"
```

To get a list of just the keys, you can use the range operator again. You can declare just one variable to only access the keys:

```
keys := []string{}

for key := range sammy {
    keys = append(keys, key)
}
fmt.Printf("%q", keys)
```

The program begins by declaring a slice to store your keys in.

The output will show only the keys of your map:

Output

```
["color" "location" "name" "animal"]
```

Again, the keys are not sorted. If you want to sort them, you use the `sort.Strings` function from the [sort](https://golang.org/pkg/sort/) package:

```
sort.Strings(keys)
```

With this function, you'll receive the following output:

Output

```
["animal" "color" "location" "name"]
```

You can use the same pattern to retrieve just the values in a map. In the next example, you pre-allocate the slice to avoid allocations, thus making the program more efficient:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark", "color": "blue

items := make([]string, len(sammy))

var i int

for _, v := range sammy {
```

```
    items[i] = v
    i++
}
fmt.Printf("%q", items)
```

First you declare a slice to store your keys in; since you know how many items you need, you can avoid potential memory allocations by defining the slice at the exact same size. You then declare your index variable. As you don't want the key, you use the _ operator, when starting your loop, to ignore the key's value. Your output would be the following:

Output

```
["ocean" "Sammy" "shark" "blue"]
```

To determine the number of items in a map, you can use the built-in `len` function:

```
sammy := map[string]string{"name": "Sammy", "animal": "shark", "color": "blue
fmt.Println(len(sammy))
```

The output displays the number of items in your map:

Output

```
4
```

Even though Go doesn't ship with convenience functions to get keys and values, it only takes a few lines of code to retrieve the keys and values when needed.

Checking Existence

Maps in Go will return the zero value for the value type of the map when the requested key is missing. Because of this, you need an alternative way to differentiate a stored zero, versus a missing key.

Let's look up a value in a map that you know doesn't exist and look at the value returned:

```
counts := map[string]int{}
fmt.Println(counts["sammy"])
```

You'll see the following output:

Output

```
0
```

Even though the key `sammy` was not in the map, Go still returned the value of `0`. This is because the value data type is an `int`, and because Go has a zero value for all variables, it returns the zero value of `0`.

In many cases, this is undesirable and would lead to a bug in your program. When looking up the value in a map, Go can return a second, optional value. This second value is a `bool` and will be `true` if the key was found, or `false` if the key was not found. In Go, this is referred to as the `ok` idiom. Even though you could name the variable that captures the second argument anything you want, in Go, you always name it `ok`:

```
count, ok := counts["sammy"]
```

If the key `sammy` exists in the `counts` map, then `ok` will be `true`. Otherwise `ok` will be false.

You can use the `ok` variable to decide what to do in your program:

```
if ok {
    fmt.Printf("Sammy has a count of %d\n", count)
} else {
    fmt.Println("Sammy was not found")
}
```

This would result in the following output:

Output

```
Sammy was not found
```

In Go, you can combine variable declaration and conditional checking with an if/else block. This allows you to use a single statement for this check:

```
if count, ok := counts["sammy"]; ok {
    fmt.Printf("Sammy has a count of %d\n", count)
} else {
    fmt.Println("Sammy was not found")
}
```

When retrieving a value from a map in Go, it's always good practice to check for its existence as well to avoid bugs in your program.

Modifying Maps

Maps are a mutable data structure, so you can modify them. Let's look at adding and deleting map items in this section.

Adding and Changing Map Items

Without using a method or function, you can add key-value pairs to maps. You do this using the maps variable name, followed by the key value in square brackets `[ ]`, and using the equal = operator to set a new value:

```
map[key] = value
```

In practice, you can see this work by adding a key-value pair to a map called `usernames`:

```
usernames := map[string]string{"Sammy": "sammy-shark", "Jamie": "mantisshrimp5

usernames["Drew"] = "squidly"
fmt.Println(usernames)
```

The output will display the new `Drew:squidly` key-value pair in the map:

Output

```
map[Drew:squidly Jamie:mantisshrimp54 Sammy:sammy-shark]
```

Because maps are returned unordered, this pair may occur anywhere in the map output. If you use the `usernames` map later in your program file, it will include the additional key-value pair.

You can also use this syntax for modifying the value assigned to a key. In this case, you reference an existing key and pass a different value to it.

Consider a map called `followers` that tracks followers of users on a given network. The user `"drew"` had a bump in followers today, so you need to update the integer value passed to the `"drew"` key. You'll use the `Println()` function to check that the map was modified:

```
followers := map[string]int{"drew": 305, "mary": 428, "cindy": 918}
followers["drew"] = 342
fmt.Println(followers)
```

Your output will show the updated value for `drew`:

Output

```
map[cindy:918 drew:342 mary:428]
```

You see that the number of followers jumped from the integer value of `305` to `342`.

You can use this method for adding key-value pairs to maps with user input. Let's write a quick program called `usernames.go` that runs on the command line and allows input from the user to add more names and associated usernames:

usernames.go

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    usernames := map[string]string{"Sammy": "sammy-shark", "Jamie": "mantissh

    for {
        fmt.Println("Enter a name:")

        var name string
        _, err := fmt.Scanln(&name)

        if err != nil {
            panic(err)
        }

        name = strings.TrimSpace(name)

        if u, ok := usernames[name]; ok {
            fmt.Printf("%q is the username of %q\n", u, name)
            continue
        }

        fmt.Printf("I don't have %v's username, what is it?\n", name)

        var username string
        _, err = fmt.Scanln(&username)

        if err != nil {
            panic(err)
        }

        username = strings.TrimSpace(username)

        usernames[name] = username
```

```
        fmt.Println("Data updated.")
    }
}
```

In `usernames.go` you first define the original map. You then set up a loop to iterate over the names. You request your user to enter a name and declare a variable to store it in. Next, you check to see if you had an error; if so, the program will exit with a panic. Because `Scanln` captures the entire input, including the carriage return, you need to remove any space from the input; you do this with the `strings.TrimSpace` function.

The `if` block checks whether the name is present in the map and prints feedback. If the name is present it then continues back to the top of the loop. If the name is not in the map, it provides feedback to the user and then will ask for a new username for the associated name. The program checks again to see if there is an error. With no error, it trims off the carriage return, assigns the username value to the name key, and then prints feedback that the data was updated.

Let's run the program on the command line:

```
go run usernames.go
```

You'll see the following output:

Output

```
Enter a name:
Sammy
"sammy-shark" is the username of "Sammy"
Enter a name:
Jesse
I don't have Jesse's username, what is it?
JOctopus
Data updated.
Enter a name:
```

When you're done testing, press `CTRL + C` to escape the program.

This shows how you can modify maps interactively. With this particular program, as soon as you exit the program with `CTRL + C` you'll lose all your data unless you implement a way to handle reading and writing files.

To summarize, you can add items to maps or modify values with the `map[key] = value` syntax.

Deleting Map Items

Just as you can add key-value pairs and change values within the map data type, you can also delete items within a map.

To remove a key-value pair from a map, you can use the built-in function `delete()`. The first argument is the map you are deleting from. The second argument is the key you are deleting:

```
delete(map, key)
```

Let's define a map of permissions:

```
permissions := map[int]string{1: "read", 2: "write", 4: "delete", 8: "create"
```

You no longer need the `modify` permission, so you'll remove it from your map. Then you'll print out the map to confirm it was removed:

```
permissions := map[int]string{1: "read", 2: "write", 4: "delete", 8: "create"
delete(permissions, 16)
fmt.Println(permissions)
```

The output will confirm the deletion:

Output

```
map[1:read 2:write 4:delete 8:create]
```

The line `delete(permissions, 16)` removes the key-value pair `16:"modify"` from the `permissions` map.

If you would like to clear a map of all of its values, you can do so by setting it equal to an empty map of the same type. This will create a new empty map to use, and the old map will be cleared from memory by the garbage collector.

Let's remove all the items within the `permissions` map:

```
permissions = map[int]string{}
fmt.Println(permissions)
```

The output shows that you now have an empty map devoid of key-value pairs:

Output

```
map[]
```

Because maps are mutable data types, they can be added to, modified, and have items removed and cleared.

Conclusion

This tutorial explored the map data structure in Go. Maps are made up of key-value pairs and provide a way to store data without relying on indexing. This allows us to retrieve values based on their meaning and relation to other data types.

Understanding Arrays and Slices in Go

Written by Gopher Guides

In Go, arrays and slices are data structures that consist of an ordered sequence of elements. These data collections are great to use when you want to work with many related values. They enable you to keep data together that belongs together, condense your code, and perform the same methods and operations on multiple values at once.

Although arrays and slices in Go are both ordered sequences of elements, there are significant differences between the two. An array in Go is a data structure that consists of an ordered sequence of elements that has its capacity defined at creation time. Once an array has allocated its size, the size can no longer be changed. A slice, on the other hand, is a variable length version of an array, providing more flexibility for developers using these data structures. Slices constitute what you would think of as arrays in other languages.

Given these differences, there are specific situations when you would use one over the other. If you are new to Go, determining when to use them can be confusing: Although the versatility of slices make them a more appropriate choice in most situations, there are specific instances in which arrays can optimize the performance of your program.

This article will cover arrays and slices in detail, which will provide you with the necessary information to make the appropriate choice when choosing between these data types. Furthermore, you'll review the most common ways to declare and work with both arrays and slices. The tutorial will first provide a description of arrays and how to manipulate them, followed by an explanation of slices and how they differ.

Arrays

Arrays are collection data structures with a set number of elements. Because the size of an array is static, the data structure only needs to allocate memory once, as opposed to a variable length data structure that must dynamically allocate memory so that it can become larger or smaller in the future. Although the fixed length of arrays can make them somewhat rigid to work with, the one-time memory allocation can increase the speed and performance of your program. Because of this, developers typically use arrays when optimizing programs in instances where the data structure will never need a variable amount of elements.

Defining an Array

Arrays are defined by declaring the size of the array in brackets `[ ]`, followed by the data type of the elements. An array in Go must have all its elements be the same data type. After the data type, you can declare the individual values of the array elements in curly brackets `{ }`.

The following is the general schema for declaring an array:

```
[capacity]data_type{element_values}
```

Note: It is important to remember that every declaration of a new array creates a distinct type. So, although `[2]int` and `[3]int` both have integer elements, their differing lengths make their data types incompatible.

If you do not declare the values of the array's elements, the default is zero-valued, which means that the elements of the array will be empty. For integers, this is represented by `0`, and for strings this is represented by an empty string.

For example, the following array `numbers` has three integer elements that do not yet have a value:

```
var numbers [3]int
```

If you printed `numbers`, you would receive the following output:

Output

```
[0 0 0]
```

If you would like to assign the values of the elements when you create the array, place the values in curly brackets. An array of strings with set values looks like this:

```
[4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}
```

You can store an array in a variable and print it out:

```
coral := [4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn c
fmt.Println(coral)
```

Running a program with the preceding lines would give you the following output:

Output

```
[blue coral staghorn coral pillar coral elkhorn coral]
```

Notice that there is no delineation between the elements in the array when it is printed, making it difficult to tell where one element ends and another begins. Because of this, it is sometimes helpful to use the `fmt.Printf` function instead, which can format strings before printing them to the screen. Provide the `%q` verb with this command to instruct the function to put quotation marks around the values:

```
fmt.Printf("%q\n", coral)
```

This will result in the following:

Output

```
["blue coral" "staghorn coral" "pillar coral" "elkhorn coral"]
```

Now each item is quoted. The `\n` verb instructs to the formatter to add a line return at the end.

With a general idea of how to declare arrays and what they consist of, you can now move on to learning how to specify elements in an array with an index number.

Indexing Arrays (and Slices)

Each element in an array (and also a slice) can be called individually through indexing. Each element corresponds to an index number, which is an `int` value starting from the index number `0` and counting up.

We will use an array in the following examples, but you could use a slice as well, since they are identical in how you index both of them.

For the array `coral`, the index breakdown looks like this:

"blue coral"	"staghorn coral"	"pillar coral"	"elkhorn coral"
0	1	2	3

The first element, the string `"blue coral"`, starts at index `0`, and the slice ends at index `3` with the element `"elkhorn coral"`.

Because each element in a slice or array has a corresponding index number, we're able to access and manipulate them in the same ways we can with other sequential data types.

Now we can call a discrete element of the slice by referring to its index number:

```
fmt.Println(coral[1])
```

Output

```
staghorn coral
```

The index numbers for this slice range from `0-3`, as shown in the previous table. So to call any of the elements individually, we would refer to the index numbers like this:

```
coral[0] = "blue coral"
coral[1] = "staghorn coral"
coral[2] = "pillar coral"
coral[3] = "elkhorn coral"
```

If we call the array `coral` with an index number of any that is greater than `3`, it will be out of range as it will not be valid:

```
fmt.Println(coral[18])
```

Output

```
panic: runtime error: index out of range
```

When indexing an array or slice, you must always use a positive number. Unlike some languages that let you index backwards with a negative number, doing that in Go will result in an error:

```
fmt.Println(coral[-1])
```

Output

```
invalid array index -1 (index must be non-negative)
```

We can concatenate string elements in an array or slice with other strings using the `+` operator:

```
fmt.Println("Sammy loves " + coral[0])
```

Output

```
Sammy loves blue coral
```

We were able to concatenate the string element at index number `0` with the string `"Sammy loves "`.

With index numbers that correspond to elements within an array or a slice, we're able to access each element discretely and work with those elements. To demonstrate this, we will next look at how to modify an element at a certain index.

Modifying Elements

We can use indexing to change elements within an array or slice by setting an index numbered element equal to a different value. This gives us greater control over the data in our slices and arrays, and will allow us to programmatically manipulate individual elements.

If we want to change the string value of the element at index `1` of the array `coral` from `"staghorn coral"` to `"foliose coral"`, we can do so like this:

```
coral[1] = "foliose coral"
```

Now when we print `coral`, the array will be different:

```
fmt.Printf("%q\n", coral)
```

Output

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral"]
```

Now that you know how to manipulate individual elements of an array or a slice, let's look at a couple of functions that will give you more flexibility when working with collection data types.

Counting Elements with `len()`

In Go, `len()` is a built-in function made to help you work with arrays and slices. Like with strings, you can calculate the length of an array or slice by using `len()` and passing in the array or slice as a parameter.

For example, to find how many elements are in the `coral` array, you would use:

```
len(coral)
```

If you print out the length for the array `coral`, you'll receive the following output:

Output

```
4
```

This gives the length of the array `4` in the `int` data type, which is correct because the array `coral` has four items:

```
coral := [4]string{"blue coral", "foliose coral", "pillar coral", "elkhorn co
```

If you create an array of integers with more elements, you could use the `len()` function on this as well:

```
numbers := [13]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
fmt.Println(len(numbers))
```

This would result in the following output:

Output

```
13
```

Although these example arrays have relatively few items, the `len()` function is especially useful when determining how many elements are in very large arrays.

Next, we will cover how to add an element to a collection data type, and demonstrate how, because of the fixed length of arrays, appending these static data types will result in an error.

Appending Elements with `append()`

`append()` is a built-in method in Go that adds elements to a collection data type. However, this method will not work when used with an array. As mentioned before, the primary way in which arrays are different from slices is that the size of an array cannot be modified. This means that while you can change the values of elements in an array, you can't make the array larger or smaller after it has been defined.

Let's consider your `coral` array:

```
coral := [4]string{"blue coral", "foliose coral", "pillar coral", "elkhorn co
```

Say you want to add the item `"black coral"` to this array. If you try to use the `append()` function with the array by typing:

```
coral = append(coral, "black coral")
```

You will receive an error as your output:

Output

```
first argument to append must be slice; have [4]string
```

To fix this, let's learn more about the slice data type, how to define a slice, and how to convert from an array to a slice.

Slices

A slice is a data type in Go that is a mutable, or changeable, ordered sequence of elements. Since the size of a slice is variable, there is a lot more flexibility when using them; when working with data collections that may need to expand or contract in the future, using a slice will ensure that your code does not run into errors when trying to manipulate the length of the collection. In most cases, this mutability is worth the possible memory re-allocation sometimes required by slices when compared to arrays. When you need to store a lot of elements or iterate over elements and you want to be able to readily modify those elements, you'll likely want to work with the slice data type.

Defining a Slice

Slices are defined by declaring the data type preceded by an empty set of square brackets (`[]`) and a list of elements between curly brackets (`{}`). You'll notice that, as opposed to arrays that require an `int` in between the brackets to declare a specific length, a slice has nothing between the brackets, representing its variable length.

Let's create a slice that contains elements of the string data type:

```
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis shrimp", "an
```

When we print out the slice, we can see the elements that are in the slice:

```
fmt.Printf("%q\n", seaCreatures)
```

This will result in the following:

Output

```
["shark" "cuttlefish" "squid" "mantis shrimp" "anemone"]
```

If you would like to create a slice of a certain length without populating the elements of the collection yet, you can use the built-in `make()` function:

```
oceans := make([]string, 3)
```

If you printed this slice, you would get:

Output

```
["" "" ""]
```

If you want to pre-allocate the memory for a certain capacity, you can pass in a third argument to `make()`:

```
oceans := make([]string, 3, 5)
```

This would make a zeroed slice with a length of `3` and a pre-allocated capacity of `5` elements.

You now know how to declare a slice. However, this does not yet solve the error we had with the `coral` array earlier. To use the `append()` function with `coral`, you will first have to learn how to slice out sections of an array.

Slicing Arrays into Slices

By using index numbers to determine beginning and endpoints, you can call a subsection of the values within an array. This is called slicing the array, and you can do this by creating a range of index numbers separated by a colon, in the form of `[first_index:second_index]`. It is important to note however, that when slicing an array, the result is a slice, not an array.

Let's say you would like to just print the middle items of the `coral` array, without the first and last element. You can do this by creating a slice starting at index `1` and ending just before index `3`:

```
fmt.Println(coral[1:3])
```

Running a program with this line would yield the following:

Output

```
[foliose coral pillar coral]
```

When creating a slice, as in `[1:3]`, the first number is where the slice starts (inclusive), and the second number is the sum of the first number and the total number of elements you would like to retrieve:

```
array[starting_index : (starting_index + length_of_slice)]
```

In this instance, you called the second element (or index 1) as the starting point, and called two elements in total. This is how the calculation would look:

```
array[1 : (1 + 2)]
```

Which is how you arrived at this notation:

```
coral[1:3]
```

If you want to set the beginning or end of the array as a starting or end point of the slice, you can omit one of the numbers in the `array[first_index:second_index]` syntax. For example, if you want to print the first three items of the array `coral` — which would be `"blue coral"`, `"foliose coral"`, and `"pillar coral"` — you can do so by typing:

```
fmt.Println(coral[:3])
```

This will print:

Output

```
[blue coral foliose coral pillar coral]
```

This printed the beginning of the array, stopping right before index `3`.

To include all the items at the end of an array, you would reverse the syntax:

```
fmt.Println(coral[1:])
```

This would give the following slice:

Output

```
[foliose coral pillar coral elkhorn coral]
```

This section discussed calling individual parts of an array by slicing out subsections. Next, you'll learn how to use slicing to convert entire arrays into slices.

Converting from an Array to a Slice

If you create an array and decide that you need it to have a variable length, you can convert it to a slice. To convert an array to a slice, use the slicing process you learned in the Slicing Arrays into Slices step of this tutorial, except this time select the entire slice by omitting both of the index numbers that would determine the endpoints:

```
coral[:]
```

Keep in mind that you can't convert the variable `coral` to a slice itself, since once a variable is defined in Go, its type can't be changed. To work around this, you can copy the entire contents of the array into a new variable as a slice:

```
coralSlice := coral[:]
```

If you printed `coralSlice`, you would receive the following output:

Output

```
[blue coral foliose coral pillar coral elkhorn coral]
```

Now, try to add the `black coral` element like in the array section, using `append()` with the newly converted slice:

```
coralSlice = append(coralSlice, "black coral")
fmt.Printf("%q\n", coralSlice)
```

This will output the slice with the added element:

Output

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black coral"]
```

We can also add more than one element in a single `append()` statement:

```
coralSlice = append(coralSlice, "antipathes", "leptopsammia")
```

Output

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black coral"
"antipathes" "leptopsammia"]
```

To combine two slices together, you can use `append()`, but you must expand the second argument to append using the `...` expansion syntax:

```
moreCoral := []string{"massive coral", "soft coral"}
coralSlice = append(coralSlice, moreCoral...)
```

Output

```
["blue coral" "foliose coral" "pillar coral" "elkhorn coral" "black coral"
"antipathes" "leptopsammia" "massive coral" "soft coral"]
```

Now that you have learned how to append an element to your slice, we will take a look at how to remove one.

Removing an Element from a Slice

Unlike other languages, Go does not provide any built-in functions to remove an element from a slice. Items need to be removed from a slice by slicing them out.

To remove an element, you must slice out the items before that element, slice out the items after that element, then append these two new slices together without the element that you wanted to remove.

If `i` is the index of the element to be removed, then the format of this process would look like the following:

```
slice = append(slice[:i], slice[i+1:]...)
```

From `coralSlice`, let’s remove the item `"elkhorn coral"`. This item is located at the index position of `3`.

```
coralSlice := []string{"blue coral", "foliose coral", "pillar coral", "elkhor

coralSlice = append(coralSlice[:3], coralSlice[4:]...)

fmt.Printf("%q\n", coralSlice)
```

Output

```
["blue coral" "foliose coral" "pillar coral" "black coral" "antipathes"
"leptopsammia" "massive coral" "soft coral"]
```

Now the element at index position `3`, the string `"elkhorn coral"`, is no longer in our slice `coralSlice`.

We can also delete a range with the same approach. Say we wanted to remove not only the item `"elkhorn coral"`, but `"black coral"` and `"antipathes"` as well. We can use a range in the expression to accomplish this:

```
coralSlice := []string{"blue coral", "foliose coral", "pillar coral", "elkhor
```

```
coralSlice = append(coralSlice[:3], coralSlice[6:]...)

fmt.Printf("%q\n", coralSlice)
```

This code will take out index 3, 4, and 5 from the slice:

Output

```
["blue coral" "foliose coral" "pillar coral" "leptopsammia" "massive coral"
"soft coral"]
```

Now that you know how to add and remove elements from a slice, let's look at how to measure the amount of data a slice can hold at any given time.

Measuring the Capacity of a Slice with `cap()`

Since slices have a variable length, the `len()`method is not the best option to determine the size of this data type. Instead, you can use the `cap()` function to learn the capacity of a slice. This will show you how many elements a slice can hold, which is determined by how much memory has already been allocated for the slice.

Note: Because the length and capacity of an array are always the same, the `cap()` function will not work on arrays.

A common use for `cap()` is to create a slice with a preset number of elements and then fill in those elements programmatically. This avoids potential unnecessary allocations that could occur by using `append()` to add elements beyond the capacity currently allocated for.

Let's take the scenario where we want to make a list of numbers, `0` through `3`. We can use `append()` in a loop to do so, or we can pre-allocate the slice first and use `cap()` to loop through to fill the values.

First, we can look at using `append()`:

```
numbers := []int{}
for i := 0; i < 4; i++ {
    numbers = append(numbers, i)
}
fmt.Println(numbers)
```

Output

```
[0 1 2 3]
```

In this example, we created a slice, and then created a `for` loop that would iterate four times. Each iteration appended the current value of the loop variable `i` into the index of the `numbers` slice. However, this could lead to unnecessary memory allocations that could slow down your program. When adding to an empty slice, each time you make a call to

append, the program checks the capacity of the slice. If the added element makes the slice exceed this capacity, the program will allocate additional memory to account for it. This creates additional overhead in your program and can result in a slower execution.

Now let's populate the slice without using `append()` by pre-allocating a certain length/capacity:

```
numbers := make([]int, 4)
for i := 0; i < cap(numbers); i++ {
    numbers[i] = i
}

fmt.Println(numbers)
```

Output

```
[0 1 2 3]
```

In this example, we used `make()` to create a slice and had it pre-allocate `4` elements. We then used the `cap()` function in the loop to iterate through each zeroed element, filling each until it reached the pre-allocated capacity. In each loop, we placed the current value of the loop variable `i` into the index of the `numbers` slice.

While the `append()` and the `cap()` strategies are both functionally equivalent, the `cap()` example avoids any additional memory allocations that would have been needed using the `append()` function.

Constructing Multidimensional Slices

You can also define slices that consist of other slices as elements, with each bracketed list enclosed inside the larger brackets of the parent slice. Collections of slices like these are called multidimensional slices. These can be thought of as depicting multidimensional coordinates; for example, a collection of five slices that are each six elements long could represent a two-dimensional grid with a horizontal length of five and a vertical height of six.

Let's examine the following multidimensional slice:

```
seaNames := [][]string{{"shark", "octopus", "squid", "mantis shrimp"}, {"Samm
```

To access an element within this slice, we will have to use multiple indices, one for each dimension of the construct:

```
fmt.Println(seaNames[1][0])
fmt.Println(seaNames[0][0])
```

In the preceding code, we first identify the element at index `0` of the slice at index `1`, then we indicate the element at index `0` of the slice at index `0`. This will yield the following:

Output

```
Sammy
shark
```

The following are the index values for the rest of the individual elements:

```
seaNames[0][0] = "shark"
seaNames[0][1] = "octopus"
seaNames[0][2] = "squid"
seaNames[0][3] = "mantis shrimp"

seaNames[1][0] = "Sammy"
seaNames[1][1] = "Jesse"
seaNames[1][2] = "Drew"
seaNames[1][3] = "Jamie"
```

When working with multidimensional slices, it is important to keep in mind that you'll need to refer to more than one index number in order to access specific elements within the relevant nested slice.

Conclusion

In this tutorial, you learned the foundations of working with arrays and slices in Go. You went through multiple exercises to demonstrate how arrays are fixed in length, whereas slices are variable in length, and discovered how this difference affects the situational uses of these data structures.

To continue studying data structures in Go, check out our article on Understanding Maps in Go, or explore the entire How To Code in Go series.

Handling Errors in Go

Written by Gopher Guides

Robust code needs to react correctly to unexpected circumstances like bad user input, faulty network connections, and failing disks. Error handling is the process of identifying when your program is in an unexpected state, and taking steps to record diagnostic information for later debugging.

Unlike other languages that require developers to handle errors with specialized syntax, errors in Go are values with the type `error` returned from functions like any other value. To handle errors in Go, we must examine these errors that functions could return, decide if an error has occurred, and take proper action to protect data and tell users or operators that the error occurred.

Creating Errors

Before we can handle errors, we need to create some first. The standard library provides two built-in functions to create errors: `errors.New` and `fmt.Errorf`. Both of these functions allow you to specify a custom error message that you can later present to your users.

`errors.New` takes a single argument—an error message as a string that you can customize to alert your users what went wrong.

Try running the following example to see an error created by `errors.New` printed to standard output:

```
package main

import (
    "errors"
    "fmt"
)
```

```go
func main() {
    err := errors.New("barnacles")
    fmt.Println("Sammy says:", err)
}
```

Output

```
Sammy says: barnacles
```

We used the `errors.New` function from the standard library to create a new error message with the string `"barnacles"` as the error message. We've followed convention here by using lowercase for the error message as the [Go Programming Language Style Guide] suggests.

Finally, we used the `fmt.Println` function to combine our error message with `"Sammy says:"`.

The `fmt.Errorf` function allows you to dynamically build an error message. Its first argument is a string containing your error message with placeholder values such as `%s` for a string and `%d` for an integer. `fmt.Errorf` interpolates the arguments that follow this formatting string into those placeholders in order:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    err := fmt.Errorf("error occurred at: %v", time.Now())
    fmt.Println("An error happened:", err)
}
```

Output

```
An error happened: Error occurred at: 2019-07-11
16:52:42.532621 -0400 EDT m=+0.000137103
```

We used the `fmt.Errorf` function to build an error message that would include the current time. The formatting string we provided to `fmt.Errorf` contains the `%v` formatting directive that tells `fmt.Errorf` to use the default formatting for the first argument provided after the formatting string. That argument will be the current time, provided by the `time.Now` function from the standard library. Similarly to the earlier example, we combine our error message with a short prefix and print the result to standard output using the `fmt.Println` function.

Handling Errors

Typically you wouldn't see an error created like this to be used immediately for no other purpose, as in the previous example. In practice, it's far more common to create an error and return it from a function when something goes wrong. Callers of that function will then use an `if` statement to see if the error was present or `nil`—an uninitialized value.

This next example includes a function that always returns an error. Notice when you run the program that it produces the same output as the previous example even though a function is returning the error this time. Declaring an error in a different location does not change the error's message.

```go
package main

import (
    "errors"
    "fmt"
)
```

```go
func boom() error {
    return errors.New("barnacles")
}

func main() {
    err := boom()

    if err != nil {
        fmt.Println("An error occurred:", err)
        return
    }
    fmt.Println("Anchors away!")
}
```

Output

```
An error occurred: barnacles
```

Here we define a function called `boom()` that returns a single `error` that we construct using `errors.New`. We then call this function and capture the error with the line `err := boom()`. Once we assign this error, we check to see if it was present with the `if err != nil` conditional. Here the conditional will always evaluate to `true`, since we are always returning an `error` from `boom()`.

This won't always be the case, so it's good practice to have logic handling cases where an error is not present (`nil`) and cases where the error is present. When the error is present, we use `fmt.Println` to print our error along with a prefix as we have done in earlier examples. Finally, we use a `return` statement to skip the execution of `fmt.Println("Anchors away!")`, since that should only execute when no error occurred.

Note: The `if err != nil` construction shown in the last example is the workhorse of error handling in the Go programming language.

Wherever a function could produce an error, it's important to use an `if` statement to check whether one occurred. In this way, idiomatic Go code naturally has its [“happy path”] logic at the first indent level, and all the “sad path” logic at the second indent level.

If statements have an optional assignment clause that can be used to help condense calling a function and handling its errors.

Run the next program to see the same output as our earlier example, but this time using a compound `if` statement to reduce some boilerplate:

```
package main

import (
    "errors"
    "fmt"
)

func boom() error {
    return errors.New("barnacles")
}

func main() {
    if err := boom(); err != nil {
        fmt.Println("An error occurred:", err)
        return
    }
    fmt.Println("Anchors away!")
}
```

Output

```
An error occurred: barnacles
```

As before, we have a function, `boom()`, that always returns an error. We assign the error returned from `boom()` to `err` as the first part of the `if` statement. In the second part of the `if` statement, following the semicolon, that `err` variable is then available. We check to see if the error was present and print our error with a short prefix string as we've done previously.

In this section, we learned how to handle functions that only return an error. These functions are common, but it's also important to be able to handle errors from functions that can return multiple values.

Returning Errors Alongside Values

Functions that return a single error value are often those that effect some stateful change, like inserting rows to a database. It's also common to write functions that return a value if they completed successfully along with a potential error if that function failed. Go permits functions to return more than one result, which can be used to simultaneously return a value and an error type.

To create a function that returns more than one value, we list the types of each returned value inside parentheses in the signature for the function. For example, a `capitalize` function that returns a `string` and an `error` would be declared using `func capitalize(name string) (string, error) {}`. The `(string, error)` part tells the Go compiler that this function will return a `string` and an `error`, in that order.

Run the following program to see the output from a function that returns both a `string` and an `error`:

```
package main

import (
    "errors"
    "fmt"
    "strings"
)
```

```go
func capitalize(name string) (string, error) {
    if name == "" {
        return "", errors.New("no name provided")
    }
    return strings.ToTitle(name), nil
}

func main() {
    name, err := capitalize("sammy")
    if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
    }

    fmt.Println("Capitalized name:", name)
}
```

Output

```
Capitalized name: SAMMY
```

We define `capitalize()` as a function that takes a string (the name to be capitalized) and returns a string and an error value. In `main()`, we call `capitalize()` and assign the two values returned from the function to the `name` and `err` variables by separating them with commas on the left-hand side of the `:=` operator. After this, we perform our `if err != nil` check as in earlier examples, printing the error to standard output using `fmt.Println` if the error was present. If no error was present, we print `Capitalized name: SAMMY`.

Try changing the string `"sammy"` in `name, err := capitalize("sammy")` to the empty string `("")` and you'll receive the error `Could not capitalize: no name provided` instead.

The `capitalize` function will return an error when callers of the function provide an empty string for the `name` parameter. When the `name` parameter is not the empty string, `capitalize()` uses `strings.ToTitle` to capitalize the `name` parameter and returns `nil` for the error value.

There are some subtle conventions that this example follows that is typical of Go code, yet not enforced by the Go compiler. When a function returns multiple values, including an error, convention requests that we return the `error` as the last item. When returning an `error` from a function with multiple return values, idiomatic Go code also will set each non-error value to a zero value. Zero values are, for example, an empty string for strings, `0` for integers, an empty struct for struct types, and `nil` for interface and pointer types, to name a few. We cover zero values in more detail in our tutorial on variables and constants.

Reducing boilerplate

Adhering to these conventions can become tedious in situations where there are many values to return from a function. We can use an anonymous function to help reduce the boilerplate. Anonymous functions are procedures assigned to variables. In contrast to the functions we have defined in earlier examples, they are only available within the functions where you declare them—this makes them perfect to act as short pieces of reusable helper logic.

The following program modifies the last example to include the length of the name that we're capitalizing. Since it has three values to return, handling errors could become cumbersome without an anonymous function to assist us:

```
package main

import (
    "errors"
    "fmt"
    "strings"
```

```
)

func capitalize(name string) (string, int, error) {
    handle := func(err error) (string, int, error) {
        return "", 0, err
    }

    if name == "" {
        return handle(errors.New("no name provided"))
    }

    return strings.ToTitle(name), len(name), nil
}

func main() {
    name, size, err := capitalize("sammy")
    if err != nil {
        fmt.Println("An error occurred:", err)
    }

    fmt.Printf("Capitalized name: %s, length: %d", name, size)
}
```

Output

```
Capitalized name: SAMMY, length: 5
```

Within `main()`, we now capture the three returned arguments from `capitalize` as `name`, `size`, and `err`, respectively. We then check to see if `capitalize` returned an `error` by checking if the `err` variable was not equal to `nil`. This is important to do before attempting to use any of the other values returned by `capitalize`, because the anonymous function, `handle`, could set those to zero values. Since no error occurred because

we provided the string `"sammy"`, we print out the capitalized name and its length.

Once again, you can try changing `"sammy"` to the empty string (`""`) to see the error case printed (`An error occurred: no name provided`).

Within `capitalize`, we define the `handle` variable as an anonymous function. It takes a single error and returns identical values in the same order as the return values of `capitalize`. `handle` sets those values to zero values and forwards the `error` passed as its argument as the final return value. Using this, we can then return any errors encountered in `capitalize` by using the `return` statement in front of the call to `handle` with the `error` as its parameter.

Remember that `capitalize` must return three values all the time, since that's how we defined the function. Sometimes we don't want to deal with all the values that a function could return. Fortunately, we have some flexibility in how we can use these values on the assignment side.

Handling Errors from Multi-Return Functions

When a function returns many values, Go requires us to assign each to a variable. In the last example, we do this by providing names for the two values returned from the `capitalize` function. These names should be separated by commas and appear on the left-hand side of the `:=` operator. The first value returned from `capitalize` will be assigned to the `name` variable, and the second value (the `error`) will be assigned to the variable `err`. Occasionally, we're only interested in the error value. You can discard any unwanted values that functions return using the special _ variable name.

In the following program, we've modified our first example involving the `capitalize` function to produce an error by passing in the empty string (`""`). Try running this program to see how we're able to examine just the error by discarding the first returned value with the _ variable:

```go
package main

import (
    "errors"
    "fmt"
    "strings"
)

func capitalize(name string) (string, error) {
    if name == "" {
        return "", errors.New("no name provided")
    }
    return strings.ToTitle(name), nil
}

func main() {
    _, err := capitalize("")
    if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
    }
    fmt.Println("Success!")
}
```

Output

```
Could not capitalize: no name provided
```

Within the `main()` function this time, we assign the capitalized name (the `string` returned first) to the underscore variable (_). At the same time, we assign the `error` returned by `capitalize` to the `err` variable. We then check if the error was present in the `if err != nil` conditional. Since we have hard-coded an empty string as an argument

to `capitalize` in the line `_, err := capitalize("")`, this conditional will always evaluate to `true`. This produces the output `"Could not capitalize: no name provided"` printed by the call to the `fmt.Println` function within the body of the `if` statement. The `return` after this will skip the `fmt.Println("Success!")`.

Conclusion

We've seen many ways to create errors using the standard library and how to build functions that return errors in an idiomatic way. In this tutorial, we've managed to successfully create various errors using the standard library `errors.New` and `fmt.Errorf` functions. In future tutorials, we'll look at how to create our own custom error types to convey richer information to users.

Creating Custom Errors in Go

Written by Gopher Guides

Go provides two methods to create errors in the standard library, `errors.New` and `fmt.Errorf`. When communicating more complicated error information to your users, or to your future self when debugging, sometimes these two mechanisms are not enough to adequately capture and report what has happened. To convey this more complex error information and attain more functionality, we can implement the standard library interface type, `error`.

The syntax for this would be as follows:

```
type error interface {
  Error() string
}
```

The `builtin` package defines `error` as an interface with a single `Error()` method that returns an error message as a string. By implementing this method, we can transform any type we define into an error of our own.

Let's try running the following example to see an implementation of the `error` interface:

```
package main

import (
    "fmt"
    "os"
)

type MyError struct{}

func (m *MyError) Error() string {
    return "boom"
}
```

```
func sayHello() (string, error) {
    return "", &MyError{}
}

func main() {
    s, err := sayHello()
    if err != nil {
        fmt.Println("unexpected error: err:", err)
        os.Exit(1)
    }
    fmt.Println("The string:", s)
}
```

We'll see the following output:

Output

```
unexpected error: err: boom
exit status 1
```

Here we've created a new empty struct type, `MyError`, and defined the `Error()` method on it. The `Error()` method returns the string `"boom"`.

Within `main()`, we call the function `sayHello` that returns an empty string and a new instance of `MyError`. Since `sayHello` will always return an error, the `fmt.Println` invocation within the body of the if statement in `main()` will always execute. We then use `fmt.Println` to print the short prefix string `"unexpected error:"` along with the instance of `MyError` held within the `err` variable.

Notice that we don't have to directly call `Error()`, since the `fmt` package is able to automatically detect that this is an implementation of `error`. It calls `Error()` transparently to get the string `"boom"` and concatenates it with the prefix string `"unexpected error: err:"`.

Collecting Detailed Information in a Custom Error

Sometimes a custom error is the cleanest way to capture detailed error information. For example, let's say we want to capture the status code for

errors produced by an HTTP request; run the following program to see an implementation of `error` that allows us to cleanly capture that information:

```
package main

import (
    "errors"
    "fmt"
    "os"
)

type RequestError struct {
    StatusCode int

    Err error
}

func (r *RequestError) Error() string {
    return fmt.Sprintf("status %d: err %v", r.StatusCode, r.Err)
}

func doRequest() error {
    return &RequestError{
        StatusCode: 503,
        Err:        errors.New("unavailable"),
    }
}

func main() {
    err := doRequest()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
```

```
    fmt.Println("success!")
}
```

We will see the following output:

Output

```
status 503: err unavailable
exit status 1
```

In this example, we create a new instance of `RequestError` and provide the status code and an error using the `errors.New` function from the standard library. We then print this using `fmt.Println` as in previous examples.

Within the `Error()` method of `RequestError`, we use the `fmt.Sprintf` function to construct a string using the information provided when the error was created.

Type Assertions and Custom Errors

The `error` interface exposes only one method, but we may need to access the other methods of `error` implementations to handle an error properly. For example, we may have several custom implementations of `error` that are temporary and can be retried—denoted by the presence of a `Temporary()` method.

Interfaces provide a narrow view into the wider set of methods provided by types, so we must use a type assertion to change the methods that view is displaying, or to remove it entirely.

The following example augments the `RequestError` shown previously to have a `Temporary()` method which will indicate whether or not callers should retry the request:

```
package main

import (
    "errors"
    "fmt"
    "net/http"
```

```
    "os"
)

type RequestError struct {
    StatusCode int

    Err error
}

func (r *RequestError) Error() string {
    return r.Err.Error()
}

func (r *RequestError) Temporary() bool {
    return r.StatusCode == http.StatusServiceUnavailable // 503
}

func doRequest() error {
    return &RequestError{
        StatusCode: 503,
        Err:        errors.New("unavailable"),
    }
}

func main() {
    err := doRequest()
    if err != nil {
        fmt.Println(err)
        re, ok := err.(*RequestError)
        if ok {
            if re.Temporary() {
                fmt.Println("This request can be tried again")
            } else {
```

```
                fmt.Println("This request cannot be tried again")
            }
        }
        os.Exit(1)
    }

    fmt.Println("success!")
}
```

We will see the following output:

Output

```
unavailable
This request can be tried again
exit status 1
```

Within `main()`, we call `doRequest()` which returns an `error` interface to us. We first print the error message returned by the `Error()` method. Next, we attempt to expose all methods from `RequestError` by using the type assertion `re, ok := err.(*RequestError)`. If the type assertion succeeded, we then use the `Temporary()` method to see if this error is a temporary error. Since the `StatusCode` set by `doRequest()` is `503`, which matches `http.StatusServiceUnavailable`, this returns `true` and causes `"This request can be tried again"` to be printed. In practice, we would instead make another request rather than printing a message.

Wrapping Errors

Commonly, an error will be generated from something outside of your program such as: a database, a network connection, etc. The error messages provided from these errors don't help anyone find the origin of the error. Wrapping errors with extra information at the beginning of an error message would provide some needed context for successful debugging.

The following example demonstrates how we can attach some contextual information to an otherwise cryptic `error` returned from some other

function:

```go
package main

import (
    "errors"
    "fmt"
)

type WrappedError struct {
    Context string
    Err     error
}

func (w *WrappedError) Error() string {
    return fmt.Sprintf("%s: %v", w.Context, w.Err)
}

func Wrap(err error, info string) *WrappedError {
    return &WrappedError{
        Context: info,
        Err:     err,
    }
}

func main() {
    err := errors.New("boom!")
    err = Wrap(err, "main")

    fmt.Println(err)
}
```

We will see the following output:

Output

```
main: boom!
```

`WrappedError` is a struct with two fields: a context message as a `string`, and an `error` that this `WrappedError` is providing more information about. When the `Error()` method is invoked, we again use `fmt.Sprintf` to print the context message, then the `error` (`fmt.Sprintf` knows to implicitly call the `Error()` method as well).

Within `main()`, we create an error using `errors.New`, and then we wrap that error using the `Wrap` function we defined. This allows us to indicate that this `error` was generated in `"main"`. Also, since our `WrappedError` is also an `error`, we could wrap other `WrappedErrors`—this would allow us to see a chain to help us track down the source of the error. With a little help from the standard library, we can even embed complete stack traces in our errors.

Conclusion

Since the `error` interface is only a single method, we've seen that we have great flexibility in providing different types of errors for different situations. This can encompass everything from communicating multiple pieces of information as part of an error all the way to implementing exponential backoff. While the error handling mechanisms in Go might on the surface seem simplistic, we can achieve quite rich handling using these custom errors to handle both common and uncommon situations.

Go has another mechanism to communicate unexpected behavior, panics. In our next article in the error handling series, we will examine panics—what they are and how to handle them.

Handling Panics in Go

Written by Gopher Guides

Errors that a program encounters fall into two broad categories: those the programmer has anticipated and those the programmer has not. The `error` interface that we have covered in our previous two articles on error handling largely deal with errors that we expect as we are writing Go programs. The `error` interface even allows us to acknowledge the rare possibility of an error occurring from function calls, so we can respond appropriately in those situations.

Panics fall into the second category of errors, which are unanticipated by the programmer. These unforeseen errors lead a program to spontaneously terminate and exit the running Go program. Common mistakes are often responsible for creating panics. In this tutorial, we'll examine a few ways that common operations can produce panics in Go, and we'll also see ways to avoid those panics. We'll also use `defer` statements along with the `recover` function to capture panics before they have a chance to unexpectedly terminate our running Go programs.

Understanding Panics

There are certain operations in Go that automatically return panics and stop the program. Common operations include indexing an array beyond its capacity, performing type assertions, calling methods on nil pointers, incorrectly using mutexes, and attempting to work with closed channels. Most of these situations result from mistakes made while programming that the compiler has no ability to detect while compiling your program.

Since panics include detail that is useful for resolving an issue, developers commonly use panics as an indication that they have made a mistake during a program's development.

Out of Bounds Panics

When you attempt to access an index beyond the length of a slice or the capacity of an array, the Go runtime will generate a panic.

The following example makes the common mistake of attempting to access the last element of a slice using the length of the slice returned by the `len` builtin. Try running this code to see why this might produce a panic:

```
package main

import (
	"fmt"
)

func main() {
	names := []string{
		"lobster",
		"sea urchin",
		"sea cucumber",
	}
	fmt.Println("My favorite sea creature is:", names[len(names)])
}
```

This will have the following output:

Output

```
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
	/tmp/sandbox879828148/prog.go:13 +0x20
```

The name of the panic's output provides a hint: `panic: runtime error: index out of range`. We created a slice with three sea creatures. We then tried to get the last element of the slice by indexing that slice with the length of the slice using the `len` builtin function. Remember that slices and arrays are zero-based; so the first element is zero and the last element in this slice is at index 2. Since we try to access the slice at the third index, 3, there is no element in the slice to return because it is beyond the bounds of the slice. The runtime has no option but to terminate and exit since we have asked it

to do something impossible. Go also can’t prove during compilation that this code will try to do this, so the compiler cannot catch this.

Notice also that the subsequent code did not run. This is because a panic is an event that completely halts the execution of your Go program. The message produced contains multiple pieces of information helpful for diagnosing the cause of the panic.

Anatomy of a Panic

Panics are composed of a message indicating the cause of the panic and a stack trace that helps you locate where in your code the panic was produced.

The first part of any panic is the message. It will always begin with the string `panic:`, which will be followed with a string that varies depending on the cause of the panic. The panic from the previous exercise has the message:

```
panic: runtime error: index out of range [3] with length 3
```

The string `runtime error:` following the `panic:` prefix tells us that the panic was generated by the language runtime. This panic is telling us that we attempted to use an index `[3]` that was out of range of the slice’s length `3`.

Following this message is the stack trace. Stack traces form a map that we can follow to locate exactly what line of code was executing when the panic was generated, and how that code was invoked by earlier code.

```
goroutine 1 [running]:
main.main()
    /tmp/sandbox879828148/prog.go:13 +0x20
```

This stack trace, from the previous example, shows that our program generated the panic from the file `/tmp/sandbox879828148/prog.go` at line number 13. It also tells us that this panic was generated in the `main()` function from the `main` package.

The stack trace is broken into separate blocks—one for each goroutine in your program. Every Go program’s execution is accomplished by one or more goroutines that can each independently and simultaneously execute parts of your Go code. Each block begins with the header `goroutine X [state]:`. The header gives the ID number of the goroutine along with the

state that it was in when the panic occurred. After the header, the stack trace shows the function that the program was executing when the panic happened, along with the filename and line number where the function executed.

The panic in the previous example was generated by an out-of-bounds access to a slice. Panics can also be generated when methods are called on pointers that are unset.

Nil Receivers

The Go programming language has pointers to refer to a specific instance of some type existing in the computer's memory at runtime. Pointers can assume the value `nil` indicating that they are not pointing at anything. When we attempt to call methods on a pointer that is `nil`, the Go runtime will generate a panic. Similarly, variables that are interface types will also produce panics when methods are called on them. To see the panics generated in these cases, try the following example:

```
package main

import (
    "fmt"
)

type Shark struct {
    Name string
}

func (s *Shark) SayHello() {
    fmt.Println("Hi! My name is", s.Name)
}

func main() {
    s := &Shark{"Sammy"}
    s = nil
```

```
    s.SayHello()
}
```

The panics produced will look like this:

Output

```
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0xffffffff addr=0x0
pc=0xdfeba]

goroutine 1 [running]:
main.(*Shark).SayHello(...)
    /tmp/sandbox160713813/prog.go:12
main.main()
    /tmp/sandbox160713813/prog.go:18 +0x1a
```

In this example, we defined a struct called `Shark`. `Shark` has one method defined on its pointer receiver called `SayHello` that will print a greeting to standard out when called. Within the body of our `main` function, we create a new instance of this `Shark` struct and request a pointer to it using the `&` operator. This pointer is assigned to the `s` variable. We then reassign the `s` variable to the value `nil` with the statement `s = nil`. Finally we attempt to call the `SayHello` method on the variable `s`. Instead of receiving a friendly message from Sammy, we receive a panic that we have attempted to access an invalid memory address. Because the `s` variable is `nil`, when the `SayHello` function is called, it tries to access the field `Name` on the `*Shark` type. Because this is a pointer receiver, and the receiver in this case is `nil`, it panics because it can't dereference a `nil` pointer.

While we have set `s` to `nil` explicitly in this example, in practice this happens less obviously. When you see panics involving `nil pointer dereference`, be sure that you have properly assigned any pointer variables that you may have created.

Panics generated from nil pointers and out-of-bounds accesses are two commonly occurring panics generated by the runtime. It is also possible to

manually generate a panic using a builtin function.

Using the `panic` Builtin Function

We can also generate panics of our own using the `panic` built-in function. It takes a single string as an argument, which is the message the panic will produce. Typically this message is less verbose than rewriting our code to return an error. Furthermore, we can use this within our own packages to indicate to developers that they may have made a mistake when using our package's code. Whenever possible, best practice is to try to return `error` values to consumers of our package.

Run this code to see a panic generated from a function called from another function:

```
package main

func main() {
    foo()
}

func foo() {
    panic("oh no!")
}
```

The panic output produced looks like:

Output

```
panic: oh no!

goroutine 1 [running]:
main.foo(...)
    /tmp/sandbox494710869/prog.go:8
main.main()
    /tmp/sandbox494710869/prog.go:4 +0x40
```

Here we define a function `foo` that calls the `panic` builtin with the string `"oh no!"`. This function is called by our `main` function. Notice how the

output has the message `panic: oh no!` and the stack trace shows a single goroutine with two lines in the stack trace: one for the `main()` function and one for our `foo()` function.

We've seen that panics appear to terminate our program where they are generated. This can create problems when there are open resources that need to be properly closed. Go provides a mechanism to execute some code always, even in the presence of a panic.

Deferred Functions

Your program may have resources that it must clean up properly, even while a panic is being processed by the runtime. Go allows you to defer the execution of a function call until its calling function has completed execution. Deferred functions run even in the presence of a panic, and are used as a safety mechanism to guard against the chaotic nature of panics. Functions are deferred by calling them as usual, then prefixing the entire statement with the `defer` keyword, as in `defer sayHello()`. Run this example to see how a message can be printed even though a panic was produced:

```
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println("hello from the deferred function!")
    }()

    panic("oh no!")
}
```

The output produced from this example will look like:

Output

```
hello from the deferred function!
panic: oh no!

goroutine 1 [running]:
main.main()

/Users/gopherguides/learn/src/github.com/gopherguides/learn//hand
le-panics/src/main.go:10 +0x55
```

Within the main function of this example, we first `defer` a call to an anonymous function that prints the message `"hello from the deferred function!"`. The `main` function then immediately produces a panic using the `panic` function. In the output from this program, we first see that the deferred function is executed and prints its message. Following this is the panic we generated in `main`.

Deferred functions provide protection against the surprising nature of panics. Within deferred functions, Go also provides us the opportunity to stop a panic from terminating our Go program using another built-in function.

Handling Panics

Panics have a single recovery mechanism—the `recover` builtin function. This function allows you to intercept a panic on its way up through the call stack and prevent it from unexpectedly terminating your program. It has strict rules for its use, but can be invaluable in a production application.

Since it is part of the `builtin` package, `recover` can be called without importing any additional packages:

```
package main

import (
	"fmt"
	"log"
)
```

```
func main() {
    divideByZero()
    fmt.Println("we survived dividing by zero!")

}

func divideByZero() {
    defer func() {
        if err := recover(); err != nil {
            log.Println("panic occurred:", err)
        }
    }()
    fmt.Println(divide(1, 0))
}

func divide(a, b int) int {
    return a / b
}
```

This example will output:

Output

```
2009/11/10 23:00:00 panic occurred: runtime error: integer divide by zero
we survived dividing by zero!
```

Our `main` function in this example calls a function we define, `divideByZero`. Within this function, we `defer` a call to an anonymous function responsible for dealing with any panics that may arise while executing `divideByZero`. Within this deferred anonymous function, we call the `recover` builtin function and assign the error it returns to a variable. If `divideByZero` is panicking, this `error` value will be set, otherwise it will be `nil`. By comparing the `err` variable against `nil`, we can detect if a panic

occurred, and in this case we log the panic using the `log.Println` function, as though it were any other `error`.

Following this deferred anonymous function, we call another function that we defined, `divide`, and attempt to print its results using `fmt.Println`. The arguments provided will cause `divide` to perform a division by zero, which will produce a panic.

In the output to this example, we first see the log message from the anonymous function that recovers the panic, followed by the message `we survived dividing by zero!`. We have indeed done this, thanks to the `recover` builtin function stopping an otherwise catastrophic panic that would terminate our Go program.

The `err` value returned from `recover()` is exactly the value that was provided to the call to `panic()`. It's therefore critical to ensure that the `err` value is only nil when a panic has not occurred.

Detecting Panics with `recover`

The `recover` function relies on the value of the error to make determinations as to whether a panic occurred or not. Since the argument to the `panic` function is an empty interface, it can be any type. The zero value for any interface type, including the empty interface, is `nil`. Care must be taken to avoid `nil` as an argument to `panic` as demonstrated by this example:

```
package main

import (
    "fmt"
    "log"
)

func main() {
    divideByZero()
    fmt.Println("we survived dividing by zero!")

}
```

```go
func divideByZero() {
    defer func() {
        if err := recover(); err != nil {
            log.Println("panic occurred:", err)
        }
    }()
    fmt.Println(divide(1, 0))
}

func divide(a, b int) int {
    if b == 0 {
        panic(nil)
    }
    return a / b
}
```

This will output:

Output

```
we survived dividing by zero!
```

This example is the same as the previous example involving `recover` with some slight modifications. The `divide` function has been altered to check if its divisor, `b`, is equal to `0`. If it is, it will generate a panic using the `panic` builtin with an argument of `nil`. The output, this time, does not include the log message showing that a panic occurred even though one was created by `divide`. This silent behavior is why it is very important to ensure that the argument to the `panic` builtin function is not `nil`.

Conclusion

We have seen a number of ways that `panic`s can be created in Go and how they can be recovered from using the `recover` builtin. While you may not necessarily use `panic` yourself, proper recovery from panics is an important step of making Go applications production-ready.

You can also explore our entire How To Code in Go series.

Importing Packages in Go

Written by Gopher Guides

There will be times when your code needs additional functionality outside of your current program. In these cases, you can use packages to make your program more sophisticated. A package represents all the files in a single directory on disk. Packages can define functions, types, and interfaces that you can reference in other Go files or packages.

This tutorial will walk you through installing, importing, and aliasing packages.

Standard Library Packages

The standard library that ships with Go is a set of packages. These packages contain many of the fundamental building blocks to write modern software. For instance, the `fmt` package contains basic functions for formatting and printing strings. The `net/http` package contains functions that allow a developer to create web services, send and retrieve data over the `http` protocol, and more.

To make use of the functions in a package, you need to access the package with an `import` statement. An `import`

statement is made up of the `import` keyword along with the name of the package.

As an example, in the Go program file `random.go` you can import the `math/rand` package to generate random numbers in this manner:

random.go

```
import "math/rand"
```

When we import a package, we are making it available in our current program as a separate namespace. This means that we will have to refer to the function in dot notation, as in `package.function`.

In practice, a function from the `math/rand` package could look like these examples:

- `rand.Int()` which calls the function to return a random integer.
- `rand.Intn()` which calls the function to return a random element from `0` up to the specified number provided.

Let's create a `for` loop to show how we will call a function of the `math/rand` package within our `random.go` program:

random.go

```go
package main

import "math/rand"

func main() {
  for i := 0; i < 10; i++ {
    println(rand.Intn(25))
  }
}
```

This program first imports the `math/rand` package on the third line, then moves into a `for` loop which that will run 10 times. Within the loop, the program will print a random integer within the range of `0` up to `25`. The integer `25` is passed to `rand.Intn()` as its parameter.

When we run the program with `go run random.go`, we'll receive 10 random integers as output. Because these are random, you'll likely get different integers each time you run the program. The output will look something like this:

Output

```
6
12
22
9
6
18
0
15
6
0
```

The integers will never go below 0 or above 24.

When importing more than one package, you can use the `()` to create a block. By using a block you can avoid repeating the `import` keyword on every line. This will make your code look cleaner:

random.go

```
import (
  "fmt"
  "math/rand"
)
```

To make use of the additional package, we can now format the output and print out the iteration that each random number was generated on during the loop:

random.go

```
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  for i := 0; i < 10; i++ {
    fmt.Printf("%d) %d\n", i, rand.Intn(25))
  }
}
```

Now, when we run our program, we'll receive output that looks like this:

Output

```
0) 6
1) 12
2) 22
3) 9
4) 6
5) 18
6) 0
7) 15
8) 6
9) 0
```

In this section, we learned how to import packages and use them to write a more sophisticated program. So far, we have only used packages from the standard library. Next, let's see how to install and use packages that are written by other developers.

Installing Packages

While the standard library ships with many great and useful packages, they are intentionally designed to be general purpose and not specific in nature. This allows developers to build their own packages on top of the standard library for their own specific needs.

The Go tool chain ships with the `go get` command. This command allows you to install third party packages to your

local development environment and use them in your program.

When using `go get` to install third party packages, it is common for a package to be referenced by its canonical path. That path can also be a path to a public project that is hosted in a code repository such as GitHub. As such, if you want to import the `flect` package, you would use the full canonical path:

```
go get github.com/gobuffalo/flect
```

The `go get` tool will find the package, on GitHub in this case, and install it into your `$GOPATH`.

For this example the code would be installed in this directory:

```
$GOPATH/src/github.com/gobuffalo/flect
```

Packages are often being updated by the original authors to address bugs or add new features. When this happens, you may want to use the latest version of that package to take advantage of the new features or resolved bug. To update a package, you can use the `-u` flag with the `go get` command:

```
go get -u github.com/gobuffalo/flect
```

This command will also have Go install the package if it is not found locally. If it is already installed, Go will attempt to update the package to the latest version.

The `go get` command always retrieves the latest version of the package available. However, there may be updates to previous versions of the package that are still newer than

you are using, and would be useful to update in your program. To retrieve that specific version of the package, you would need to use a Package Management tool, such as Go Modules.

As of Go 1.11, Go Modules are used to manage what version of the package you want imported. The topic of package management is beyond the scope of this article, but you can read more about it on the Go Modules GitHub page.

Aliasing Imported Packages

You may want to change a package name if you have a local package already named the same as a third party package you are using. When this happens, aliasing your import is the best way to handle the collision. You can modify the names of packages and their functions within Go by putting an `alias` name in front of the imported package.

The construction of this statement looks like this:

```
import another_name "package"
```

In this example, modify the name of the `fmt` package in the `random.go` program file. We'll change the package name of `fmt` to `f` in order to abbreviate it. Our modified program will look like this:

random.go

```
package main

import (
 f "fmt"
  "math/rand"
)

func main() {
  for i := 0; i < 10; i++ {
    f.Printf("%d) %d\n", i, rand.Intn(25))
  }
}
```

Within the program, we now refer to the `Printf` function as `f.Printf` rather than `fmt.Printf`.

While other languages favor aliasing a package for ease of use later in the program, Go does not. For instance, aliasing the `fmt` package to `f` would not be consistent with the style guide.

When renaming imports to avoid a name collision, you should aim to rename the most local or project specific import. For instance, if you had a local package called `strings`, and you also needed to import the system package called `strings`, you would favor renaming your

local package over the system package. Whenever possible, it's best to avoid name collision altogether.

In this section, we learned how we can alias an import to avoid colliding with another import in our program. It is important to remember that readability and clarity of your program is important, so you should only use aliasing to make the code more readable or when you need to avoid a naming collision.

Formatting Imports

By formatting imports, you can sort the packages into a specific order that will make your code more consistent. Additionally, this will prevent random commits from taking place when the only thing that changes is the sort order of the imports. Since formatting imports will prevent random commits, this will prevent unnecessary code churn and confusing code reviews.

Most editors will format imports for you automatically, or will let you configure your editor to use `goimports`. It is considered standard practice to use `goimports` in your editor, as trying to manually maintain the sort order of your imports can be tedious and prone to errors. Additionally, if any style changes are made, `goimports` will be updated to reflect those style changes. This ensures that you, and anyone that works on your code, will have consistent styling in your import blocks.

Here is what an example import block may look like before formatting:

```go
import (
  "fmt"
  "os"
  "github.com/digital/ocean/godo"
  "github.com/sammy/foo"
  "math/rand"
  "github.com/sammy/bar"
)
```

Running the `goimport` tool (or with most editors that have it installed, saving the file will run it for you), you will now have the following format:

```go
import (
  "fmt"
  "math/rand"
  "os"

  "github.com/sammy/foo"
  "github.com/sammy/bar"

  "github.com/digital/ocean/godo"
)
```

Notice that it groups all standard library packages first and then groups third party packages together with blank

lines. This makes is easier to read and understand what packages are being used.

In this section we learned that using `goimports` will keep all of our import blocks properly formatted, and prevent unnecessary code churn between developers working on the same files.

Conclusion

When we import packages we're able to call functions that are not built in to Go. Some packages are part of the standard library that installs with Go, and some we will install through `go get`.

Making use of packages allows us to make our programs more robust and powerful as we're leveraging existing code. We can also create our own packages for ourselves and for other programmers to use in future programs.

How To Write Packages in Go

Written by Gopher Guides

A package is made up of Go files that live in the same directory and have the same package statement at the beginning. You can include additional functionality from packages to make your programs more sophisticated. Some packages are available through the Go Standard Library and are therefore installed with your Go installation. Others can be installed with Go's `go get` command. You can also build your own Go packages by creating Go files in the same directory across which you want to share code by using the necessary package statement.

This tutorial will guide you through writing Go packages for use within other programming files.

Prerequisites

- Set up a Go programming environment following one of the tutorials from the How To Install and Set Up a Local Programming Environment for Go series. Create your Go Workspace following Step 5 in the Local Programming Environment tutorials. To follow the example and naming conventions in this article, read the first section Writing and Importing Packages.

- To deepen your knowledge of the GOPATH, read our article Understanding the GOPATH.

Writing and Importing Packages

Writing a package is just like writing any other Go file. Packages can contain definitions of functions, types, and variables that can then be used in other Go programs.

Before we create a new package, we need to be in our Go workspace. This is typically under our `gopath`. For the example, in this tutorial we will call the package `greet`. To do this, we've created a directory called `greet` in our `gopath` under our project space. If our organization were `gopherguides`, and we wanted to create the `greet` package under the organization while using Github as our code repository, then our directory would look like this:

```
└── $GOPATH
    └── src
        └── github.com
            └── gopherguides
```

The `greet` directory is within the `gopherguides` directory:

```
└── $GOPATH
    └── src
        └── github.com
            └── gopherguides
                └── greet
```

Finally, we can add the first file in our directory. It is considered common practice that the `primary` or `entry point` file in a package is named after the name of the directory. In this case, we would create a file called `greet.go` inside the `greet` directory:

```
└── $GOPATH
    └── src
        └── github.com
            └── gopherguides
                └── greet
                    └── greet.go
```

With the file created, we can begin to write our code that we want to reuse or share across projects. In this case, we will create a function called `Hello` that prints out `Hello World`.

Open your `greet.go` file in your text editor and add the following code:

greet.go

```go
package greet

import "fmt"

func Hello() {
    fmt.Println("Hello, World!")
}
```

Let's break this first file down. The first line of each file needs the name of the `package` that you are working in. Since you're in the `greet` package, you use the `package` keyword followed by the name of the package:

```
package greet
```

This will tell the compiler to treat everything in the file as being part of the `greet` package.

Next you declare any other packages you need to use with the `import` statement. You're only using one in this file —the `fmt` package:

```
import "fmt"
```

Lastly, you create the function `Hello`. It will use the `fmt` package to print out `Hello, World!`:

```
func Hello() {
    fmt.Println("Hello, World!")
}
```

Now that you've written the `greet` package, you can use it in any other package you create. Let's create a new package in which you'll use your `greet` package.

You're going to create a package called `example`, which means you need a directory called `example`. Create this package in your `gopherguides` organization, so the directory structure looks like so:

```
└── $GOPATH
    └── src
        └── github.com
```

```
                └── gopherguides
                        └── example
```

Now that you have your directory for your new package, you can create the entry point file. Because this is going to be an executable program, it is considered best practice to name the entry point file `main.go`:

```
└── $GOPATH
    └── src
        └── github.com
            └── gopherguides
                └── example
                    └── main.go
```

In your text editor, open `main.go` and add the following code to call the `greet` package:

main.go

```go
package main

import "github.com/gopherguides/greet"

func main() {
    greet.Hello()
}
```

Because you're importing a package, you need to call the function by referencing the package name in dot notation.

Dot notation is the practice of putting a period `.` between the name of the package you are using and the resource within that package that you want to use. For instance, in your `greet` package, you have the `Hello` function as a resource. If you want to call that resource, you use the dot notation of `greet.Hello()`.

Now, you can open your terminal and run the program on the command line:

```
go run main.go
```

When you do, you'll receive the following output:

Output

```
Hello, World!
```

To see how you can use variables in a package, let's add a variable definition in your `greet.go` file:

greet.go

```
package greet

import "fmt"

var Shark = "Sammy"

func Hello() {
    fmt.Println("Hello, World!")
}
```

Next, open your `main.go` file and add the following highlighted line to call the variable from `greet.go` in a `fmt.Println()` function:

main.go

```
package main

import (
    "fmt"

    "github.com/gopherguides/greet"
)

func main() {
    greet.Hello()

    fmt.Println(greet.Shark)
}
```

Once you run the program again:

```
go run main.go
```

You'll receive the following output:

Output

```
Hello, World!
Sammy
```

Finally, let’s also define a type in the `greet.go` file. You’ll create the type `Octopus` with `name` and `color` fields, and a function that will print out the fields when called:

greet.go

```
package greet

import "fmt"

var Shark = "Sammy"

type Octopus struct {
    Name  string
    Color string
}

func (o Octopus) String() string {
    return fmt.Sprintf("The octopus's name is %q and is the color %s.", o.Name, o.Color)
}

func Hello() {
    fmt.Println("Hello, World!")
}
```

Open `main.go` to create an instance of that type at the end of the file:

main.go

```
package main

import (
	"fmt"

	"github.com/gopherguides/greet"
)

func main() {
	greet.Hello()

	fmt.Println(greet.Shark)

	oct := greet.Octopus{
		Name:  "Jesse",
		Color: "orange",
	}

	fmt.Println(oct.String())
}
```

Once you've created an instance of `Octopus` type with `oct := greet.Octopus`, you can access the functions and fields of the type within the `main.go` file's namespace. This lets you write `oct.String()` on the last line without invoking `greet`. You could also, for example, call one of the

types fields such as `oct.Color` without referencing the name of the `greet` package.

The `String` method on the `Octopus` type uses the `fmt.Sprintf` function to create a sentence, and `returns` the result, a string, to the caller (in this case, your main program).

When you run the program, you'll receive the following output:

```
go run main.go
```

Output

```
Hello, World!
Sammy
The octopus's name is "Jesse" and is the color
orange.
```

By creating the `String` method on `Octopus`, you now have a reusable way to print out information about your custom type. If you want to change the behavior of this method in the future, you only have to edit this one method.

Exported Code

You may have noticed that all of the declarations in the `greet.go` file you called were capitalized. Go does not have the concept of `public`, `private`, or `protected` modifiers like other languages do. External visibility is controlled by

capitalization. Types, variables, functions, and so on, that start with a capital letter are available, publicly, outside the current package. A symbol that is visible outside its package is considered to be `exported`.

If you add a new method to `Octopus` called `reset`, you can call it from within the `greet` package, but not from your `main.go` file, which is outside the `greet` package:

greet.go

```
package greet

import "fmt"

var Shark = "Sammy"

type Octopus struct {
    Name  string
    Color string
}

func (o Octopus) String() string {
    return fmt.Sprintf("The octopus's name is %q
and is the color %s.", o.Name, o.Color)
}

func (o *Octopus) reset() {
    o.Name = ""
    o.Color = ""
}

func Hello() {
    fmt.Println("Hello, World!")
}
```

If you try to call `reset` from the `main.go` file:

main.go

```
package main

import (
    "fmt"

    "github.com/gopherguides/greet"
)

func main() {
    greet.Hello()

    fmt.Println(greet.Shark)

    oct := greet.Octopus{
        Name:  "Jesse",
        Color: "orange",
    }

    fmt.Println(oct.String())

    oct.reset()
}
```

You’ll receive the following compilation error:

Output

```
oct.reset undefined (cannot refer to unexported
field or method greet.Octopus.reset)
```

To `export` the `reset` functionality from `Octopus`, capitalize the `R` in `reset`:

greet.go

```
package greet

import "fmt"

var Shark = "Sammy"

type Octopus struct {
    Name  string
    Color string
}

func (o Octopus) String() string {
    return fmt.Sprintf("The octopus's name is %q
and is the color %s.", o.Name, o.Color)
}

func (o *Octopus) Reset() {
    o.Name = ""
    o.Color = ""
}

func Hello() {
    fmt.Println("Hello, World!")
}
```

As a result you can call `Reset` from your other package without getting an error:

main.go

```go
package main

import (
    "fmt"

    "github.com/gopherguides/greet"
)

func main() {
    greet.Hello()

    fmt.Println(greet.Shark)

    oct := greet.Octopus{
        Name:  "Jesse",
        Color: "orange",
    }

    fmt.Println(oct.String())

    oct.Reset()

    fmt.Println(oct.String())
}
```

Now if you run the program:

```
go run main.go
```

You will receive the following output:

Output

```
Hello, World!
Sammy
The octopus's name is "Jesse" and is the color
orange
The octopus's name is "" and is the color .
```

By calling `Reset`, you cleared out all the information in the `Name` and `Color` fields. When you call the `String` method, it will print nothing where `Name` and `Color` normally appear because the fields are now empty.

Conclusion

Writing a Go package is the same as writing any other Go file, but placing it in another directory allows you to isolate the code to be reused elsewhere. This tutorial covered how to write definitions within a package, demonstrated how to make use of those definitions within another Go programming file, and explained the options for where to keep the package in order to access it.

Understanding Package Visibility in Go

Written by Gopher Guides

When creating a package in Go, the end goal is usually to make the package accessible for other developers to use, either in higher order packages or whole programs. By importing the package, your piece of code can serve as the building block for other, more complex tools. However, only certain packages are available for importing. This is determined by the visibility of the package.

Visibility in this context means the file space from which a package or other construct can be referenced. For example, if we define a variable in a function, the visibility (scope) of that variable is only within the function in which it was defined. Similarly, if you define a variable in a package, you can make it visible to just that package, or allow it to be visible outside the package as well.

Carefully controlling package visibility is important when writing ergonomic code, especially when accounting for future changes that you may want to make to your package. If you need to fix a bug, improve performance, or change functionality, you'll want to make the change in a way that won't break the code of anyone using your package. One way to minimize breaking changes is to allow access only to the parts of your package that are needed for it to be used properly. By limiting access, you can make changes internally to your package with less of a chance of affecting how other developers are using your package.

In this article, you will learn how to control package visibility, as well as how to protect parts of your code that should only be used inside your package. To do this, we will create a basic logger to log and debug messages, using packages with varying degrees of item visibility.

Prerequisites

To follow the examples in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment. This tutorial will use the following file structure:

```
.
├── bin
│
└── src
```

```
    └── github.com
        └── gopherguides
```

Exported and Unexported Items

Unlike other program languages like Java and [Python](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-python-3) that use access modifiers such as `public`, `private`, or `protected` to specify scope, Go determines if an item is `exported` and `unexported` through how it is declared. Exporting an item in this case makes it `visible` outside the current package. If it's not exported, it is only visible and usable from within the package it was defined.

This external visibility is controlled by capitalizing the first letter of the item declared. All declarations, such as `Types`, `Variables`, `Constants`, `Functions`, etc., that start with a capital letter are visible outside the current package.

Let's look at the following code, paying careful attention to capitalization:

greet.go

```
package greet

import "fmt"

var Greeting string

func Hello(name string) string {
    return fmt.Sprintf(Greeting, name)
}
```

This code declares that it is in the `greet` package. It then declares two symbols, a variable called `Greeting`, and a function called `Hello`. Because they both start with a capital letter, they are both `exported` and available to any outside program. As stated earlier, crafting a package that limits access will allow for better API design and make it easier to update your package internally without breaking anyone's code that is depending on your package.

Defining Package Visibility

To give a closer look at how package visibility works in a program, let's create a `logging` package, keeping in mind what we want to make visible outside our package and what we won't make visible. This logging package will be responsible for logging any of our program messages to the console. It will also look at what level we are

logging at. A level describes the type of log, and is going to be one of three statuses: `info`, `warning`, or `error`.

First, within your `src` directory, let's create a directory called `logging` to put our logging files in:

```
mkdir logging
```

Move into that directory next:

```
cd logging
```

Then, using an editor like nano, create a file called `logging.go`:

```
nano logging.go
```

Place the following code in the `logging.go` file we just created:

logging/logging.go

```go
package logging

import (
    "fmt"
    "time"
)

var debug bool

func Debug(b bool) {
    debug = b
}

func Log(statement string) {
    if !debug {
        return
    }

    fmt.Printf("%s %s\n", time.Now().Format(time.RFC3339), statement)
}
```

The first line of this code declared a package called `logging`. In this package, there are two `exported` functions: `Debug` and `Log`. These functions can be called by any other package that imports the `logging` package. There is also a private variable called `debug`. This variable is only accessible from within the `logging` package. It is important to note that while the function `Debug` and the variable `debug` both have the

same spelling, the function is capitalized and the variable is not. This makes them distinct declarations with different scopes.

Save and quit the file.

To use this package in other areas of our code, we can import it into a new package. We'll create this new package, but we'll need a new directory to store those source files in first.

Let's move out of the `logging` directory, create a new directory called `cmd`, and move into that new directory:

```
cd ..
mkdir cmd
cd cmd
```

Create a file called `main.go` in the `cmd` directory we just created:

```
nano main.go
```

Now we can add the following code:

cmd/main.go

```go
package main

import "github.com/gopherguides/logging"

func main() {
    logging.Debug(true)

    logging.Log("This is a debug statement...")
}
```

We now have our entire program written. However, before we can run this program, we'll need to also create a couple of configuration files for our code to work properly. Go uses Go Modules to configure package dependencies for importing resources. Go modules are configuration files placed in your package directory that tell the compiler where to import packages from. While learning about modules is beyond the scope of this article, we can write just a couple lines of configuration to make this example work locally.

Open the following `go.mod` file in the `cmd` directory:

```
nano go.mod
```

Then place the following contents in the file:

go.mod

```
module github.com/gopherguides/cmd

replace github.com/gopherguides/logging => ../logging
```

The first line of this file tells the compiler that the `cmd` package has a file path of `github.com/gopherguides/cmd`. The second line tells the compiler that the package `github.com/gopherguides/logging` can be found locally on disk in the `../logging` directory.

We'll also need a `go.mod` file for our `logging` package. Let's move back into the `logging` directory and create a `go.mod` file:

```
cd ../logging
nano go.mod
```

Add the following contents to the file:

go.mod

```
module github.com/gopherguides/logging
```

This tells the compiler that the `logging` package we created is actually the `github.com/gopherguides/logging` package. This makes it possible to import the package in our `main` package with the following line that we wrote earlier:

cmd/main.go

```
package main

import "github.com/gopherguides/logging"

func main() {
    logging.Debug(true)

    logging.Log("This is a debug statement...")
}
```

You should now have the following directory structure and file layout:

```
├── cmd
│   ├── go.mod
│   └── main.go
└── logging
```

```
├── go.mod
└── logging.go
```

Now that we have all the configuration completed, we can run the `main` program from the `cmd` package with the following commands:

```
cd ../cmd
go run main.go
```

You will get output similar to the following:

Output

```
2019-08-28T11:36:09-05:00 This is a debug statement...
```

The program will print out the current time in RFC 3339 format followed by whatever statement we sent to the logger. [RFC 3339](https://tools.ietf.org/html/rfc3339) is a time format that was designed to represent time on the internet and is commonly used in log files.

Because the `Debug` and `Log` functions are exported from the logging package, we can use them in our `main` package. However, the `debug` variable in the `logging` package is not exported. Trying to reference an unexported declaration will result in a compile-time error.

Add the following highlighted line to `main.go`:

cmd/main.go

```
package main

import "github.com/gopherguides/logging"

func main() {
    logging.Debug(true)

    logging.Log("This is a debug statement...")

    fmt.Println(logging.debug)
}
```

Save and run the file. You will receive an error similar to the following:

Output

```
. . .
./main.go:10:14: cannot refer to unexported name logging.debug
```

Now that we have seen how `exported` and `unexported` items in packages behave, we will next look at how `fields` and `methods` can be exported from `structs`.

Visibility Within Structs

While the visibility scheme in the logger we built in the last section may work for simple programs, it shares too much state to be useful from within multiple packages. This is because the exported variables are accessible to multiple packages that could modify the variables into contradictory states. Allowing the state of your package to be changed in this way makes it hard to predict how your program will behave. With the current design, for example, one package could set the `Debug` variable to `true`, and another could set it to `false` in the same instance. This would create a problem since both packages that are importing the `logging` package are affected.

We can make the logger isolated by creating a struct and then hanging methods off of it. This will allow us to create an `instance` of a logger to be used independently in each package that consumes it.

Change the `logging` package to the following to refactor the code and isolate the logger:

logging/logging.go

```go
package logging

import (
    "fmt"
    "time"
)

type Logger struct {
    timeFormat string
    debug      bool
}

func New(timeFormat string, debug bool) *Logger {
    return &Logger{
        timeFormat: timeFormat,
        debug:      debug,
    }
}

func (l *Logger) Log(s string) {
    if !l.debug {
        return
    }
    fmt.Printf("%s %s\n", time.Now().Format(l.timeFormat), s)
}
```

In this code, we created a `Logger` struct. This struct will house our unexported state, including the time format to print out and the `debug` variable setting of `true` or `false`. The `New` function sets the initial state to create the logger with, such as the time format and debug state. It then stores the values we gave it internally to the unexported variables `timeFormat` and `debug`. We also created a method called `Log` on the `Logger` type that takes a statement we want to print out. Within the `Log` method is a reference to its local method variable `l` to get access back to its internal fields such as `l.timeFormat` and `l.debug`.

This approach will allow us to create a `Logger` in many different packages and use it independently of how the other packages are using it.

To use it in another package, let's alter `cmd/main.go` to look like the following:

cmd/main.go

```go
package main

import (
    "time"

    "github.com/gopherguides/logging"
)

func main() {
    logger := logging.New(time.RFC3339, true)

    logger.Log("This is a debug statement...")
}
```

Running this program will give you the following output:

Output

```
2019-08-28T11:56:49-05:00 This is a debug statement...
```

In this code, we created an instance of the logger by calling the exported function `New`. We stored the reference to this instance in the `logger` variable. We can now call `logging.Log` to print out statements.

If we try to reference an unexported field from the `Logger` such as the `timeFormat` field, we will receive a compile-time error. Try adding the following highlighted line and running `cmd/main.go`:

cmd/main.go

```
package main

import (
    "time"

    "github.com/gopherguides/logging"
)

func main() {
    logger := logging.New(time.RFC3339, true)

    logger.Log("This is a debug statement...")

    fmt.Println(logger.timeFormat)
}
```

This will give the following error:

Output

```
. . .
cmd/main.go:14:20: logger.timeFormat undefined (cannot refer to
unexported field or method timeFormat)
```

The compiler recognizes that `logger.timeFormat` is not exported, and therefore can't be retrieved from the `logging` package.

Visibility Within Methods

In the same way as struct fields, methods can also be exported or unexported.

To illustrate this, let's add leveled logging to our logger. Leveled logging is a means of categorizing your logs so that you can search your logs for specific types of events. The levels we will put into our logger are:

- The `info` level, which represents information type events that inform the user of an action, such as `Program started`, or `Email sent`. These help us debug and track parts of our program to see if expected behavior is happening.

- The `warning` level. These types of events identify when something unexpected is happening that is not an error, like `Email failed to send, retrying`. They help us see parts of our program that aren't going as smoothly as we expected them to.
- The `error` level, which means the program encountered a problem, like `File not found`. This will often result in the program's operation failing.

You may also desire to turn on and off certain levels of logging, especially if your program isn't performing as expected and you'd like to debug the program. We'll add this functionality by changing the program so that when `debug` is set to `true`, it will print all levels of messages. Otherwise, if it's `false`, it will only print error messages.

Add leveled logging by making the following changes to `logging/logging.go`:

logging/logging.go

```go
package logging

import (
    "fmt"
    "strings"
    "time"
)

type Logger struct {
    timeFormat string
    debug      bool
}

func New(timeFormat string, debug bool) *Logger {
    return &Logger{
        timeFormat: timeFormat,
        debug:      debug,
    }
}

func (l *Logger) Log(level string, s string) {
    level = strings.ToLower(level)
    switch level {
    case "info", "warning":
        if l.debug {
            l.write(level, s)
        }
    default:
        l.write(level, s)
    }
}

func (l *Logger) write(level string, s string) {
    fmt.Printf("[%s] %s %s\n", level, time.Now().Format(l.timeFormat), s)
}
```

In this example, we introduced a new argument to the `Log` method. We can now pass in the `level` of the log message. The `Log` method determines what level of message it is. If it's an `info` or `warning` message, and the `debug` field is `true`, then it writes the message. Otherwise it ignores the message. If it is any other level, like `error`, it will write out the message regardless.

Most of the logic for determining if the message is printed out exists in the `Log` method. We also introduced an unexported method called `write`. The `write` method is what actually outputs the log message.

We can now use this leveled logging in our other package by changing `cmd/main.go` to look like the following:

cmd/main.go

```go
package main

import (
    "time"

    "github.com/gopherguides/logging"
)

func main() {
    logger := logging.New(time.RFC3339, true)

    logger.Log("info", "starting up service")
    logger.Log("warning", "no tasks found")
    logger.Log("error", "exiting: no work performed")

}
```

Running this will give you:

Output

```
[info] 2019-09-23T20:53:38Z starting up service
[warning] 2019-09-23T20:53:38Z no tasks found
[error] 2019-09-23T20:53:38Z exiting: no work performed
```

In this example, `cmd/main.go` successfully used the exported `Log` method.

We can now pass in the `level` of each message by switching `debug` to `false`:

main.go

```go
package main

import (
	"time"

	"github.com/gopherguides/logging"
)

func main() {
	logger := logging.New(time.RFC3339, false)

	logger.Log("info", "starting up service")
	logger.Log("warning", "no tasks found")
	logger.Log("error", "exiting: no work performed")

}
```

Now we will see that only the `error` level messages print:

Output

```
[error] 2019-08-28T13:58:52-05:00 exiting: no work performed
```

If we try to call the `write` method from outside the `logging` package, we will receive a compile-time error:

main.go

```
package main

import (
    "time"

    "github.com/gopherguides/logging"
)

func main() {
    logger := logging.New(time.RFC3339, true)

    logger.Log("info", "starting up service")
    logger.Log("warning", "no tasks found")
    logger.Log("error", "exiting: no work performed")

    logger.write("error", "log this message...")
}
```

Output

```
cmd/main.go:16:8: logger.write undefined (cannot refer to unexported
field or method logging.(*Logger).write)
```

When the compiler sees that you are trying to reference something from another package that starts with a lowercase letter, it knows that it is not exported, and therefore throws a compiler error.

The logger in this tutorial illustrates how we can write code that only exposes the parts we want other packages to consume. Because we control what parts of the package are visible outside the package, we are now able to make future changes without affecting any code that depends on our package. For example, if we wanted to only turn off `info` level messages when `debug` is false, you could make this change without affecting any other part of your API. We could also safely make changes to the log message to include more information, such as the directory the program was running from.

Conclusion

This article showed how to share code between packages while also protecting the implementation details of your package. This allows you to export a simple API that

will seldom change for backwards compatibility, but will allow for changes privately in your package as needed to make it work better in the future. This is considered a best practice when creating packages and their corresponding APIs.

To learn more about packages in Go, check out our Importing Packages in Go and How To Write Packages in Go articles, or explore our entire How To Code in Go series.

How To Write Conditional Statements in Go

Written by Gopher Guides

Conditional statements are part of every programming language. With conditional statements, we can have code that sometimes runs and at other times does not run, depending on the conditions of the program at that time.

When we fully execute each statement of a program, we are not asking the program to evaluate specific conditions. By using conditional statements, programs can determine whether certain conditions are being met and then be told what to do next.

Let’s look at some examples where we would use conditional statements:

- If the student receives over 65% on her test, report that her grade passes; if not, report that her grade fails.
- If he has money in his account, calculate interest; if he doesn’t, charge a penalty fee.
- If they buy 10 oranges or more, calculate a discount of 5%; if they buy fewer, then don’t.

Through evaluating conditions and assigning code to run based on whether or not those conditions are met, we are writing conditional code.

This tutorial will take you through writing conditional statements in the Go programming language.

If Statements

We will start with the `if` statement, which will evaluate whether a statement is true or false, and run code only in the case that the statement is true.

In a plain text editor, open a file and write the following code:

grade.go

```
package main

import "fmt"

func main() {
    grade := 70

    if grade >= 65 {
        fmt.Println("Passing grade")
    }
}
```

With this code, we have the variable `grade` and are giving it the integer value of `70`. We are then using the `if` statement to evaluate whether or not the variable grade is greater than or equal (`>=`) to `65`. If it does meet this condition, we are telling the program to print out the string `Passing grade`.

Save the program as `grade.go` and run it in a local programming environment from a terminal window with the command `go run grade.go`.

In this case, the grade of 70 does meet the condition of being greater than or equal to 65, so you will receive the following output once you run the program:

Output

```
Passing grade
```

Let's now change the result of this program by changing the value of the `grade` variable to `60`:

grade.go

```
package main

import "fmt"

func main() {
    grade := 60

    if grade >= 65 {
        fmt.Println("Passing grade")
    }
}
```

When we save and run this code, we will receive no output because the condition was not met and we did not tell the program to execute another statement.

To give one more example, let us calculate whether a bank account balance is below 0. Let's create a file called `account.go` and write the following program:

account.go

```go
package main

import "fmt"

func main() {
    balance := -5

    if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be charged
    }
}
```

When we run the program with `go run account.go`, we'll receive the following output:

Output

```
Balance is below 0, add funds now or you will be charged a penalty.
```

In the program we initialized the variable `balance` with the value of `-5`, which is less than 0. Since the balance met the condition of the `if` statement (`balance < 0`), once we save and run the code, we will receive the string output. Again, if we change the balance to 0 or a positive number, we will receive no output.

Else Statements

It is likely that we will want the program to do something even when an `if` statement evaluates to false. In our grade example, we will want output whether the grade is passing or failing.

To do this, we will add an `else` statement to the grade condition above that is constructed like this:

grade.go

```
package main

import "fmt"

func main() {
    grade := 60

    if grade >= 65 {
        fmt.Println("Passing grade")
    } else {
        fmt.Println("Failing grade")
    }
}
```

Since the grade variable has the value of `60`, the `if` statement evaluates as false, so the program will not print out `Passing grade`. The `else` statement that follows tells the program to do something anyway.

When we save and run the program, we'll receive the following output:

Output

```
Failing grade
```

If we then rewrite the program to give the grade a value of `65` or higher, we will instead receive the output `Passing grade`.

To add an `else` statement to the bank account example, we rewrite the code like this:

account.go

```go
package main

import "fmt"

func main() {
    balance := 522

    if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be charged
    } else {
        fmt.Println("Your balance is 0 or above.")
    }
}
```

Output

```
Your balance is 0 or above.
```

Here, we changed the `balance` variable value to a positive number so that the `else` statement will print. To get the first `if` statement to print, we can rewrite the value to a negative number.

By combining an `if` statement with an `else` statement, you are constructing a two-part conditional statement that will tell the computer to execute certain code whether or not the `if` condition is met.

Else if Statements

So far, we have presented a Boolean option for conditional statements, with each `if` statement evaluating to either true or false. In many cases, we will want a program that evaluates more than two possible outcomes. For this, we will use an else if statement, which is written in Go as `else if`. The `else if` or else if statement looks like the `if` statement and will evaluate another condition.

In the bank account program, we may want to have three discrete outputs for three different situations:

- The balance is below 0
- The balance is equal to 0
- The balance is above 0

The `else if` statement will be placed between the `if` statement and the `else` statement as follows:

account.go

```go
package main

import "fmt"

func main() {
    balance := 522

    if balance < 0 {
        fmt.Println("Balance is below 0, add funds now or you will be charged
    } else if balance == 0 {
        fmt.Println("Balance is equal to 0, add funds soon.")
    } else {
        fmt.Println("Your balance is 0 or above.")
    }
}
```

Now, there are three possible outputs that can occur once we run the program: - If the variable `balance` is equal to `0` we will receive the output from the `else if` statement (`Balance is equal to 0, add funds soon.`) - If the variable `balance` is set to a positive number, we will receive the output from the `else` statement (`Your balance is 0 or above.`). - If the variable `balance` is set to a negative number, the output will be the string from the `if` statement (`Balance is below 0, add funds now or you will be charged a penalty`).

What if we want to have more than three possibilities, though? We can do this by writing more than one `else if` statement into our code.

In the `grade.go` program, let's rewrite the code so that there are a few letter grades corresponding to ranges of numerical grades:

- 90 or above is equivalent to an A grade
- 80-89 is equivalent to a B grade
- 70-79 is equivalent to a C grade
- 65-69 is equivalent to a D grade
- 64 or below is equivalent to an F grade

To run this code, we will need one `if` statement, three `else if` statements, and an `else` statement that will handle all failing cases.

Let's rewrite the code from the preceding example to have strings that print out each of the letter grades. We can keep our `else` statement the same.

grade.go

```go
package main

import "fmt"

func main() {
    grade := 60

    if grade >= 90 {
        fmt.Println("A grade")
    } else if grade >= 80 {
        fmt.Println("B grade")
    } else if grade >= 70 {
        fmt.Println("C grade")
    } else if grade >= 65 {
        fmt.Println("D grade")
    } else {
        fmt.Println("Failing grade")
    }
}
```

Since `else if` statements will evaluate in order, we can keep our statements pretty basic. This program is completing the following steps:

1. If the grade is greater than 90, the program will print `A grade`, if the grade is less than 90, the program will continue to the next statement…
2. If the grade is greater than or equal to 80, the program will print `B grade`, if the grade is 79 or less, the program will continue to the next statement…
3. If the grade is greater than or equal to 70, the program will print `C grade`, if the grade is 69 or less, the program will continue to the next statement…
4. If the grade is greater than or equal to 65, the program will print `D grade`, if the grade is 64 or less, the program will continue to the next statement…
5. The program will print `Failing grade` because all of the above conditions were not met.

Nested If Statements

Once you are feeling comfortable with the `if`, `else if`, and `else` statements, you can move on to nested conditional statements. We can use nested `if` statements for situations where we want to check for a secondary condition if the first condition executes as true.

For this, we can have an if-else statement inside of another if-else statement. Let's look at the syntax of a nested `if` statement:

```
if statement1 { // outer if statement
    fmt.Println("true")

    if nested_statement { // nested if statement
        fmt.Println("yes")
    } else { // nested else statement
        fmt.Println("no")
    }

} else { // outer else statement
    fmt.Println("false")
}
```

A few possible outputs can result from this code:

- If `statement1` evaluates to true, the program will then evaluate whether the `nested_statement` also evaluates to true. If both cases are true, the output will be:

```
[secondary_label Output]
true
yes
```

- If, however, `statement1` evaluates to true, but `nested_statement` evaluates to false, then the output will be:

```
[secondary_label Output]
true
no
```

- And if `statement1` evaluates to false, the nested if-else statement will not run, so the `else` statement will run alone, and the output will be:

```
[secondary_label Output]
false
```

We can also have multiple `if` statements nested throughout our code:

```
if statement1 { // outer if
    fmt.Println("hello world")

    if nested_statement1 { // first nested if
        fmt.Println("yes")
```

```
    } else if nested_statement2 { // first nested else if
        fmt.Println("maybe")

    } else { // first nested else
        fmt.Println("no")
    }

} else if statement2 { // outer else if
    fmt.Println("hello galaxy")

    if nested_statement3 { // second nested if
        fmt.Println("yes")
    } else if nested_statement4 { // second nested else if
        fmt.Println("maybe")
    } else { // second nested else
        fmt.Println("no")
    }

} else { // outer else
    statement("hello universe")
}
```

In this code, there is a nested `if` statement inside each `if` statement in addition to the `else if` statement. This will allow for more options within each condition.

Let's look at an example of nested `if` statements with our `grade.go` program. We can check for whether a grade is passing first (greater than or equal to 65%), then evaluate which letter grade the numerical grade should be equivalent to. If the grade is not passing, though, we do not need to run through the letter grades, and instead can have the program report that the grade is failing. Our modified code with the nested `if` statement will look like this:

grade.go

```go
package main

import "fmt"

func main() {
    grade := 92
    if grade >= 65 {
        fmt.Print("Passing grade of: ")

        if grade >= 90 {
            fmt.Println("A")

        } else if grade >= 80 {
            fmt.Println("B")

        } else if grade >= 70 {
            fmt.Println("C")

        } else if grade >= 65 {
            fmt.Println("D")
        }

    } else {
        fmt.Println("Failing grade")
    }
}
```

If we run the code with the variable `grade` set to the integer value `92`, the first condition is met, and the program will print out `Passing grade of:`. Next, it will check to see if the grade is greater than or equal to 90, and since this condition is also met, it will print out `A`.

If we run the code with the `grade` variable set to `60`, then the first condition is not met, so the program will skip the nested `if` statements and move down to the `else` statement, with the program printing out `Failing grade`.

We can of course add even more options to this, and use a second layer of nested if statements. Perhaps we will want to evaluate for grades of A+, A and A- separately. We can do so by first checking if the grade is passing, then checking to see if the grade is 90 or above, then checking to see if the grade is over 96 for an A+:

grade.go

```
...
if grade >= 65 {
    fmt.Print("Passing grade of: ")

    if grade >= 90 {
        if grade > 96 {
            fmt.Println("A+")

        } else if grade > 93 && grade <= 96 {
            fmt.Println("A")

        } else {
            fmt.Println("A-")
        }
...
```

In this code, for a `grade` variable set to `96`, the program will run the following:

1. Check if the grade is greater than or equal to 65 (true)
2. Print out `Passing grade of:`
3. Check if the grade is greater than or equal to 90 (true)
4. Check if the grade is greater than 96 (false)
5. Check if the grade is greater than 93 and also less than or equal to 96 (true)
6. Print `A`
7. Leave these nested conditional statements and continue with remaining code

The output of the program for a grade of 96 therefore looks like this:

Output

```
Passing grade of: A
```

Nested `if` statements can provide the opportunity to add several specific levels of conditions to your code.

Conclusion

By using conditional statements like the `if` statement, you will have greater control over what your program executes. Conditional statements tell the program to evaluate whether a certain condition is being met. If the condition is met it will execute specific code, but if it is not met the program will continue to move down to other code.

To continue practicing conditional statements, try using different operators to gain more familiarity with conditional statements.

How To Write Switch Statements in Go

Written by Gopher Guides

Conditional statements give programmers the ability to direct their programs to take some action if a condition is true and another action if the condition is false. Frequently, we want to compare some variable against multiple possible values, taking different actions in each circumstance. It's possible to make this work using `if` statements alone. Writing software, however, is not only about making things work but also communicating your intention to your future self and other developers. `switch` is an alternative conditional statement useful for communicating actions taken by your Go programs when presented with different options.

Everything we can write with the switch statement can also be written with `if` statements. In this tutorial, we'll look at a few examples of what the switch statement can do, the `if` statements it replaces, and where it's most appropriately applied.

Structure of Switch Statements

Switch is commonly used to describe the actions taken by a program when a variable is assigned specific values. The following example demonstrates how we would accomplish this using `if` statements:

```
package main

import "fmt"

func main() {
    flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}

    for _, flav := range flavors {
        if flav == "strawberry" {
            fmt.Println(flav, "is my favorite!")
            continue
        }

        if flav == "vanilla" {
            fmt.Println(flav, "is great!")
```

```
            continue
        }

        if flav == "chocolate" {
            fmt.Println(flav, "is great!")
            continue
        }

        fmt.Println("I've never tried", flav, "before")
    }
}
```

This will generate the following output:

Output

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
I've never tried banana before
```

Within `main`, we define a slice of ice-cream flavors. We then use a `for loop` to iterate through them. We use three `if` statements to print out different messages indicating preferences for different ice-cream flavors. Each `if` statement must use the `continue` statement to stop execution of the `for` loop so that the default message at the end is not printed for the preferred ice-cream flavors.

As we add new ice-cream preferences, we have to keep adding `if` statements to handle the new cases. Duplicated messages, as in the case of `"vanilla"` and `"chocolate"`, must have duplicated `if` statements. To future readers of our code (ourselves included), the repetitive nature of the `if` statements obscures the important part of what they are doing—comparing the variable against multiple values and taking different actions. Also, our fallback message is set apart from the conditionals, making it appear unrelated. The `switch` statement can help us organize this logic better.

The `switch` statement begins with the `switch` keyword and is followed, in its most basic form, with some variable to perform comparisons against. This is followed by a pair of curly braces (`{}`) where multiple case clauses can appear. Case clauses describe the actions your Go program should take when the variable provided to the switch statement equals the value referenced by the case clause.

The following example converts the previous example to use a `switch` instead of multiple `if` statements:

```
package main

import "fmt"

func main() {
    flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}

    for _, flav := range flavors {
        switch flav {
        case "strawberry":
            fmt.Println(flav, "is my favorite!")
        case "vanilla", "chocolate":
            fmt.Println(flav, "is great!")
        default:
            fmt.Println("I've never tried", flav, "before")
        }
    }
}
```

The output is the same as before:

Output

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
I've never tried banana before
```

We've once again defined a slice of ice-cream flavors in `main` and used the `range` statement to iterate over each flavor. This time, however, we've used a `switch` statement that will examine the `flav` variable. We use two `case` clauses to indicate preferences. We no longer need `continue` statements as only one `case` clause will be executed by the `switch` statement. We're also able to combine the duplicated logic of the `"chocolate"` and `"vanilla"` conditionals by separating each with a comma in the declaration of the `case` clause. The `default` clause serves as our catch-all clause. It will run for any flavors that we haven't accounted for in the

body of the `switch` statement. In this case, `"banana"` will cause `default` to execute, printing the message `I've never tried banana before`.

This simplified form of `switch` statements addresses the most common use for them: comparing a variable against multiple alternatives. It also provides conveniences for us where we want to take the same action for multiple different values and some other action when none of the listed conditions are met by using the provided `default` keyword.

When this simplified form of `switch` proves too limiting, we can use a more general form of `switch` statement.

General Switch Statements

`switch` statements are useful for grouping collections of more complicated conditionals to show that they are somehow related. This is most commonly used when comparing some variable against a range of values, rather than specific values as in the earlier example. The following example implements a guessing game using `if` statements that could benefit from a `switch` statement:

```
package main

import (
	"fmt"
	"math/rand"
	"time"
)

func main() {
	rand.Seed(time.Now().UnixNano())
	target := rand.Intn(100)

	for {
		var guess int
		fmt.Print("Enter a guess: ")
		_, err := fmt.Scanf("%d", &guess)
		if err != nil {
			fmt.Println("Invalid guess: err:", err)
			continue
		}
```

```
        if guess > target {
            fmt.Println("Too high!")
            continue
        }

        if guess < target {
            fmt.Println("Too low!")
            continue
        }

        fmt.Println("You win!")
        break
    }
}
```

The output will vary depending on the random number selected and how well you play the game. Here is the output from one example session:

Output

```
Enter a guess: 10
Too low!
Enter a guess: 15
Too low!
Enter a guess: 18
Too high!
Enter a guess: 17
You win!
```

Our guessing game needs a random number to compare guesses against, so we use the `rand.Intn` function from the `math/rand` package. To make sure we get different values for `target` each time we play the game, we use `rand.Seed` to randomize the random number generator based on the current time. The argument `100` to `rand.Intn` will give us a number in the range 0–100. We then use a `for` loop to begin collecting guesses from the player.

The `fmt.Scanf` function gives us a means to read user input into a variable of our choosing. It takes a format string verb that converts the user's input into the type we expect. `%d` here means we expect an `int`, and we pass the address of the `guess` variable so that `fmt.Scanf` is able to set that variable. After handling any

parsing errors we then use two `if` statements to compare the user's guess to the `target` value. The `string` that they return, along with `bool`, controls the message displayed to the player and whether the game will exit.

These `if` statements obscure the fact that the range of values that the variable is being compared against are all related in some way. It can also be difficult, at a glance, to tell if we missed some part of the range. The next example refactors the previous example to use a `switch` statement instead:

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    target := rand.Intn(100)

    for {
        var guess int
        fmt.Print("Enter a guess: ")
        _, err := fmt.Scanf("%d", &guess)
        if err != nil {
            fmt.Println("Invalid guess: err:", err)
            continue
        }

        switch {
        case guess > target:
            fmt.Println("Too high!")
        case guess < target:
            fmt.Println("Too low!")
        default:
            fmt.Println("You win!")
            return
        }
    }
}
```

This will generate output similar to the following:

Output

```
Enter a guess: 25
Too low!
Enter a guess: 28
Too high!
Enter a guess: 27
You win!
```

In this version of the guessing game, we've replaced the block of `if` statements with a `switch` statement. We omit the expression argument to `switch` because we are only interested in using `switch` to collect conditionals together. Each `case` clause contains a different expression comparing `guess` against `target`. Similar to the first time we replaced `if` statements with `switch`, we no longer need `continue` statements since only one `case` clause will be executed. Finally, the `default` clause handles the case where `guess == target` since we have covered all other possible values with the other two `case` clauses.

In the examples that we've seen so far, exactly one case statement will be executed. Occasionally, you may wish to combine the behaviors of multiple `case` clauses. `switch` statements provide another keyword for achieving this behavior.

Fallthrough

Sometimes you will want to reuse the code that another `case` clause contains. In these cases, it's possible to ask Go to run the body of the next `case` clause listed using the `fallthrough` keyword. This next example modifies our earlier ice cream flavor example to more accurately reflect our enthusiasm for strawberry ice cream:

```go
package main

import "fmt"

func main() {
    flavors := []string{"chocolate", "vanilla", "strawberry", "banana"}

    for _, flav := range flavors {
        switch flav {
```

```
        case "strawberry":
            fmt.Println(flav, "is my favorite!")
            fallthrough
        case "vanilla", "chocolate":
            fmt.Println(flav, "is great!")
        default:
            fmt.Println("I've never tried", flav, "before")
        }
    }
}
```

We will see this output:

Output

```
chocolate is great!
vanilla is great!
strawberry is my favorite!
strawberry is great!
I've never tried banana before
```

As we've seen previously, we define a slice of `string` to represent flavors and iterate through this using a `for` loop. The `switch` statement here is identical to the one we've seen before, but with the addition of the `fallthrough` keyword at the end of the `case` clause for `"strawberry"`. This will cause Go to run the body of `case "strawberry":`, first printing out the string `strawberry is my favorite!`. When it encounters `fallthrough` it will run the body of the next `case` clause. This will cause the body of `case "vanilla", "chocolate":` to run, printing `strawberry is great!`.

The `fallthrough` keyword is not used often by Go developers. Usually, the code reuse realized by using `fallthrough` can be better obtained by defining a function with the common code. For these reasons, using `fallthrough` is generally discouraged.

Conclusion

`switch` statements help us convey to other developers reading our code that a set of comparisons are somehow related to each other. They make it much easier to add different behavior when a new case is added in the future and make it possible to ensure that anything we forgot is handled properly as well with `default`

clauses. The next time you find yourself writing multiple `if` statements that all involve the same variable, try rewriting it with a `switch` statement—you'll find it easier to rework when it comes time to consider some other alternative value.

If you'd like to learn more about the Go programming language, check out the entire How To Code in Go series.

How To Construct For Loops in Go

Written by Gopher Guides

In computer programming, a loop is a code structure that loops around to repeatedly execute a piece of code, often until some condition is met. Using loops in computer programming allows you to automate and repeat similar tasks multiple times. Imagine if you had a list of files that you needed to process, or if you wanted to count the number of lines in an article. You would use a loop in your code to solve these types of problems.

In Go, a `for` loop implements the repeated execution of code based on a loop counter or loop variable. Unlike other programming languages that have multiple looping constructs such as `while`, `do`, etc., Go only has the `for` loop. This serves to make your code clearer and more readable, since you do not have to worry with multiple strategies to achieve the same looping construct. This enhanced readability and decreased cognitive load during development will also make your code less prone to error than in other languages.

In this tutorial, you will learn how Go's `for` loop works, including the three major variations of its use. We'll start by showing how to create different types of `for` loops, followed by how to loop through sequential data types in Go. We'll end by explaining how to use nested loops.

Declaring ForClause and Condition Loops

In order to account for a variety of use cases, there are three distinct ways to create `for` loops in Go, each with their own capabilities. These are to create a `for` loop with a Condition, a ForClause, or a RangeClause. In this section, we will explain how to declare and use the ForClause and Condition variants.

Let's look at how we can use a `for` loop with the ForClause first.

A ForClause loop is defined as having an initial statement, followed by a condition, and then a post statement. These are arranged in the following syntax:

```
for [ Initial Statement ] ; [ Condition ] ; [ Post Statement ] {
    [Action]
}
```

To explain what the preceding components do, let's look at a `for` loop that increments through a specified range of values using the ForClause syntax:

```go
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

Let's break this loop down and identify each part.

The first part of the loop is `i := 0`. This is the initial statement:

```go
for i := 0; i < 5; i++ {
    fmt.Println(i)
```

```
}
```

It states that we are declaring a variable called `i`, and setting the initial value to `0`.

Next is the condition:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

In this condition, we stated that while `i` is less than the value of `5`, the loop should continue looping.

Finally, we have the post statement:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

In the post statement, we increment the loop variable `i` up by one each time an iteration occurs using the `i++` increment operator.

When we run this program, the output looks like this:

Output

```
0
1
2
3
4
```

The loop ran 5 times. Initially, it set `i` to `0`, and then checked to see if `i` was less than `5`. Since the value of `i` was less than `5`, the loop executed and the action of `fmt.Println(i)` was executed. After the loop finished, the post statement of `i++` was called, and the value of `i` was incremented by 1.

Note: Keep in mind that in programming we tend to begin at index 0, so that is why although 5 numbers are printed out, they range from 0-4.

We aren't limited to starting at 0 or ending at a specified value. We can assign any value to our initial statement, and also stop at any value in our post statement. This allows us to create any desired range to loop through:

```
for i := 20; i < 25; i++ {
    fmt.Println(i)
}
```

Here, the iteration goes from 20 (inclusive) to 25 (exclusive), so the output looks like this:

Output

```
20
21
22
23
24
```

We can also use our post statement to increment at different values. This is similar to `step` in other languages:

First, let's use a post statement with a positive value:

```
for i := 0; i < 15; i += 3 {
    fmt.Println(i)
}
```

In this case, the `for` loop is set up so that the numbers from 0 to 15 print out, but at an increment of 3, so that only every third number is printed, like so:

Output

```
0
3
6
9
12
```

We can also use a negative value for our post statement argument to iterate backwards, but we'll have to adjust our initial statement and condition arguments accordingly:

```
for i := 100; i > 0; i -= 10 {
    fmt.Println(i)
}
```

Here, we set `i` to an initial value of `100`, use the condition of `i < 0` to stop at `0`, and the post statement decrements the value by 10 with the `-=` operator. The loop begins at `100` and ends at `0`, decreasing by 10 with each iteration. We can see this occur in the output:

Output

```
100
90
80
70
60
50
40
30
20
10
```

You can also exclude the initial statement and the post statement from the `for` syntax, and only use the condition. This is what is known as a Condition loop:

```
i := 0
for i < 5 {
    fmt.Println(i)
    i++
}
```

This time, we declared the variable `i` separately from the `for` loop in the preceding line of code. The loop only has a condition clause that checks to see if `i` is less than `5`. As long as the condition evaluates to `true`, the loop will continue to iterate.

Sometimes you may not know the number of iterations you will need to complete a certain task. In that case, you can omit all statements, and use the `break` keyword to exit execution:

```
for {
    if someCondition {
        break
    }
    // do action here
}
```

An example of this may be if we are reading from an indeterminately sized structure like a buffer and we don't know when we will be done reading:

buffer.go

```
package main

import (
    "bytes"
    "fmt"
    "io"
)

func main() {
    buf := bytes.NewBufferString("one\ntwo\nthree\nfour\n")

    for {
        line, err := buf.ReadString('\n')
        if err != nil {
            if err == io.EOF {

                fmt.Print(line)
                break
            }
            fmt.Println(err)
            break
        }
        fmt.Print(line)
    }
}
```

In the preceding code, `buf :=bytes.NewBufferString("one\ntwo\nthree\nfour\n")` declares a buffer with some data. Because we don't know when the buffer will finish reading, we create a `for` loop with no clause. Inside the `for` loop, we use `line, err := buf.ReadString('\n')` to read a line from the buffer and check to see if there was an error reading from the buffer. If there was, we address the error, and [use the break keyword to exit the for loop](). With these `break` points, you do not need to include a condition to stop the loop.

In this section, we learned how to declare a ForClause loop and use it to iterate through a known range of values. We also learned how to use a Condition loop to iterate until a specific condition was met. Next, we'll learn how the RangeClause is used for iterating through sequential data types.

Looping Through Sequential Data Types with RangeClause

It is common in Go to use `for` loops to iterate over the elements of sequential or collection data types like slices, arrays, and strings. To make it easier to do so, we can use a `for` loop with RangeClause syntax. While you can loop through sequential data types using the ForClause syntax, the RangeClause is cleaner and easier to read.

Before we look at using the RangeClause, let's look at how we can iterate through a slice by using the ForClause syntax:

main.go

```
package main

import "fmt"

func main() {
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled",
"bullhead", "requiem"}

    for i := 0; i < len(sharks); i++ {
        fmt.Println(sharks[i])
    }
}
```

Running this will give the following output, printing out each element of the slice:

Output

```
hammerhead
great white
dogfish
frilled
bullhead
requiem
```

Now, let's use the RangeClause to perform the same set of actions:

main.go

```
package main

import "fmt"

func main() {
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled",
"bullhead", "requiem"}

    for i, shark := range sharks {
        fmt.Println(i, shark)
    }
}
```

In this case, we are printing out each item in the list. Though we used the variables `i` and `shark`, we could have called the variable any other [valid variable name]() and we would get the same output:

Output

```
0 hammerhead
1 great white
2 dogfish
3 frilled
4 bullhead
5 requiem
```

When using `range` on a slice, it will always return two values. The first value will be the index that the current iteration of the loop is in, and the second is the value at that index. In this case, for the first iteration, the index was `0`, and the value was `hammerhead`.

Sometimes, we only want the value inside the slice elements, not the index. If we change the preceding code to only print out the value however, we will receive a compile time error:

main.go

```
package main

import "fmt"

func main() {
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled",
"bullhead", "requiem"}

    for i, shark := range sharks {
        fmt.Println(shark)
    }
}
```

Output

```
src/range-error.go:8:6: i declared and not used
```

Because `i` is declared in the `for` loop, but never used, the compiler will respond with the error of `i declared and not used`. This is the same error that you will receive in Go any time you declare a variable and don't use it.

Because of this, Go has the blank identifier which is an underscore (_). In a `for` loop, you can use the blank identifier to ignore any value returned from the `range` keyword. In this case, we want to ignore the index, which is the first argument returned.

main.go

```
package main

import "fmt"

func main() {
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled",
"bullhead", "requiem"}

    for _, shark := range sharks {
        fmt.Println(shark)
    }
}
```

Output

```
hammerhead
great white
dogfish
frilled
bullhead
requiem
```

This output shows that the `for` loop iterated through the slice of strings, and printed each item from the slice without the index.

You can also use `range` to add items to a list:

main.go

```go
package main

import "fmt"

func main() {
    sharks := []string{"hammerhead", "great white", "dogfish", "frilled",
"bullhead", "requiem"}

    for range sharks {
        sharks = append(sharks, "shark")
    }

    fmt.Printf("%q\n", sharks)
}
```

Output

```
['hammerhead', 'great white', 'dogfish', 'frilled', 'bullhead', 'requiem',
'shark', 'shark', 'shark', 'shark', 'shark', 'shark']
```

Here, we have added a placeholder string of `"shark"` for each item of the length of the `sharks` slice.

Notice that we didn't have to use the blank identifier `_` to ignore any of the return values from the `range` operator. Go allows us to leave out the entire declaration portion of the `range` statement if we don't need to use either of the return values.

We can also use the `range` operator to fill in values of a slice:

main.go

```
package main

import "fmt"

func main() {
    integers := make([]int, 10)
    fmt.Println(integers)

    for i := range integers {
        integers[i] = i
    }

    fmt.Println(integers)
}
```

In this example, the slice `integers` is initialized with ten empty values, but the `for` loop sets all the values in the list like so:

Output

```
[0 0 0 0 0 0 0 0 0 0]
[0 1 2 3 4 5 6 7 8 9]
```

The first time we print the value of the slice `integers`, we see all zeros. Then we iterate through each index and set the value to the current index. Then when we print the value of `integers` a second time, showing that they all now have a value of `0` through `9`.

We can also use the `range` operator to iterate through each character in a string:

main.go

```
package main

import "fmt"

func main() {
    sammy := "Sammy"

    for _, letter := range sammy {
        fmt.Printf("%c\n", letter)
    }
}
```

Output

```
S
a
m
m
y
```

When iterating through a map, `range` will return both the key and the value:

main.go

```
package main

import "fmt"

func main() {
    sammyShark := map[string]string{"name": "Sammy", "animal": "shark", "colo

    for key, value := range sammyShark {
        fmt.Println(key + ": " + value)
    }
}
```

Output

```
color: blue
location: ocean
name: Sammy
animal: shark
```

Note: It is important to note that the order in which a map returns is random. Each time you run this program you may get a different result.

Now that we have learned how to iterate over sequential data with `range` `for` loops, let's look at how to use loops inside of loops.

Nested For Loops

Loops can be nested in Go, as they can with other programming languages. Nesting is when we have one construct inside of another. In this case, a nested loop is a loop that occurs within another loop. These can be useful when you would like to have a looped action performed on every element of a data set.

Nested loops are structurally similar to nested `if` statements. They are constructed like so:

```
for {
    [Action]
    for {
        [Action]
    }
}
```

The program first encounters the outer loop, executing its first iteration. This first iteration triggers the inner, nested loop, which then runs to completion. Then the program returns back to the top of the outer loop, completing the second iteration and again triggering the nested loop. Again, the nested loop runs to completion, and the program returns back to the top of the outer loop until the sequence is complete or a break or other statement disrupts the process.

Let's implement a nested `for` loop so we can take a closer look. In this example, the outer loop will iterate through a slice of integers called `numList`, and the inner loop will iterate through a slice of strings called `alphaList`.

main.go

```go
package main

import "fmt"

func main() {
    numList := []int{1, 2, 3}
    alphaList := []string{"a", "b", "c"}

    for _, i := range numList {
        fmt.Println(i)
        for _, letter := range alphaList {
            fmt.Println(letter)
        }
    }
}
```

When we run this program, we'll receive the following output:

Output

```
1
a
b
c
2
a
b
c
3
a
b
c
```

The output illustrates that the program completes the first iteration of the outer loop by printing `1`, which then triggers completion of the inner loop, printing `a`, `b`, `c` consecutively. Once the inner loop has completed, the program returns to the top of the outer loop, prints `2`, then again prints the inner loop in its entirety (`a`, `b`, `c`), etc.

Nested `for` loops can be useful for iterating through items within slices composed of slices. In a slice composed of slices, if we use just one `for` loop, the program will output each internal list as an item:

main.go

```go
package main

import "fmt"

func main() {
    ints := [][]int{
        []int{0, 1, 2},
        []int{-1, -2, -3},
        []int{9, 8, 7},
    }

    for _, i := range ints {
        fmt.Println(i)
    }
}
```

Output

```
[0 1 2]
[-1 -2 -3]
[9 8 7]
```

In order to access each individual item of the internal slices, we'll implement a nested `for` loop:

main.go

```go
package main

import "fmt"

func main() {
    ints := [][]int{
        []int{0, 1, 2},
        []int{-1, -2, -3},
        []int{9, 8, 7},
    }

    for _, i := range ints {
        for _, j := range i {
            fmt.Println(j)
        }
    }
}
```

Output

```
0
1
2
-1
-2
-3
9
8
7
```

When we use a nested `for` loop here, we are able to iterate over the individual items contained in the slices.

Conclusion

In this tutorial we learned how to declare and use `for` loops to solve for repetitive tasks in Go. We also learned the three different variations of a `for` loop and when to use them. To learn more about `for` loops and how to control the flow of them, read Using Break and Continue Statements When Working with Loops in Go.

Using Break and Continue Statements When Working with Loops in Go

Written by Gopher Guides

Using for loops in Go allow you to automate and repeat tasks in an efficient manner.

Learning how to control the operation and flow of loops will allow for customized logic in your program. You can control your loops with the `break` and `continue` statements.

Break Statement

In Go, the `break` statement terminates execution of the current loop. A `break` is almost always paired with a conditional `if` statement.

Let's look at an example that uses the `break` statement in a `for` loop:

break.go

```go
package main

import "fmt"

func main() {
	for i := 0; i < 10; i++ {
		if i == 5 {
			fmt.Println("Breaking out of loop")
			break // break here
		}
		fmt.Println("The value of i is", i)
	}
	fmt.Println("Exiting program")
}
```

This small program creates a `for` loop that will iterate while `i` is less than `10`.

Within the `for` loop, there is an `if` statement. The `if` statement tests the condition of `i` to see if the value is less than `5`. If the value of `i` is not equal to `5`, the loop continues and prints out the value of `i`. If the value of `i` is equal to `5`, the loop will execute the `break` statement, print that it is `Breaking out of loop`, and stop executing the loop. At the end of the program we print out `Exiting program` to signify that we have exited the loop.

When we run this code, our output will be the following:

Output

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
Breaking out of loop
Exiting program
```

This shows that once the integer `i` is evaluated as equivalent to 5, the loop breaks, as the program is told to do so with the `break` statement.

Nested Loops

It is important to remember that the `break` statement will only stop the execution of the inner most loop it is called in. If you have a nested set of loops, you will need a break for each loop if desired.

nested.go

```go
package main

import "fmt"

func main() {
	for outer := 0; outer < 5; outer++ {
		if outer == 3 {
			fmt.Println("Breaking out of outer loop")
			break // break here
		}
		fmt.Println("The value of outer is", outer)
		for inner := 0; inner < 5; inner++ {
			if inner == 2 {
				fmt.Println("Breaking out of inner loop")
				break // break here
			}
			fmt.Println("The value of inner is", inner)
		}
	}
	fmt.Println("Exiting program")
}
```

In this program, we have two loops. While both loops iterate 5 times, each has a conditional `if` statement with a `break` statement. The outer loop will break if the value of `outer` equals `3`. The inner loop will break if the value of `inner` is `2`.

If we run the program, we can see the output:

Output

```
The value of outer is 0
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
The value of outer is 1
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
The value of outer is 2
The value of inner is 0
The value of inner is 1
Breaking out of inner loop
Breaking out of outer loop
Exiting program
```

Notice that each time the inner loop breaks, the outer loop does not break. This is because `break` will only break the inner most loop it is called from.

We have seen how using `break` will stop the execution of a loop. Next, let's look at how we can continue the iteration of a loop.

Continue Statement

The `continue` statement is used when you want to skip the remaining portion of the loop, and return to the top of the loop and continue a new iteration.

As with the `break` statement, the `continue` statement is commonly used with a conditional `if` statement.

Using the same `for` loop program as in the preceding Break Statement section, we'll use a `continue` statement rather than a `break` statement:

continue.go

```
package main

import "fmt"

func main() {
    for i := 0; i < 10; i++ {
        if i == 5 {
            fmt.Println("Continuing loop")
            continue // break here
        }
        fmt.Println("The value of i is", i)
    }
    fmt.Println("Exiting program")
}
```

The difference in using the `continue` statement rather than a `break` statement is that our code will continue despite the disruption when the variable `i` is evaluated as equivalent to `5`. Let's look at our output:

Output

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
Continuing loop
The value of i is 6
The value of i is 7
The value of i is 8
The value of i is 9
Exiting program
```

Here we see that the line `The value of i is 5` never occurs in the output, but the loop continues after that point to print lines for the numbers 6-10 before leaving the loop.

You can use the `continue` statement to avoid deeply nested conditional code, or to optimize a loop by eliminating frequently occurring cases that you would like to reject.

The `continue` statement causes a program to skip certain factors that come up within a loop, but then continue through the rest of the loop.

Conclusion

The `break` and `continue` statements in Go will allow you to use `for` loops more effectively in your code.

How To Define and Call Functions in Go

Written by Gopher Guides

A function is a section of code that, once defined, can be reused. Functions are used to make your code easier to understand by breaking it into small, understandable tasks that can be used more than once throughout your program.

Go ships with a powerful standard library that has many predefined functions. Ones that you are probably already familiar with from the fmt package are:

- `fmt.Println()` which will print objects to standard out (most likely your terminal).
- `fmt.Printf()` which will allow you to format your printed output.

Function names include parentheses and may include parameters.

In this tutorial, we'll go over how to define your own functions to use in your coding projects.

Defining a Function

Let's start with turning the classic "Hello, World!" program into a function.

We'll create a new text file in our text editor of choice, and call the program `hello.go`. Then, we'll define the function.

A function is defined by using the `func` keyword. This is then followed by a name of your choosing and a set of parentheses that hold any parameters the function will take (they can be empty). The lines of function code are enclosed in curly brackets `{}`.

In this case, we'll define a function named `hello()`:

hello.go

```go
func hello() {}
```

This sets up the initial statement for creating a function.

From here, we'll add a second line to provide the instructions for what the function does. In this case, we'll be printing `Hello, World!` to the console:

hello.go

```go
func hello() {
    fmt.Println("Hello, World!")
}
```

Our function is now fully defined, but if we run the program at this point, nothing will happen since we didn't call the function.

So, inside of our `main()` function block, let's call the function with `hello()`:

hello.go

```
package main

import "fmt"

func main() {
    hello()
}

func hello() {
    fmt.Println("Hello, World!")
}
```

Now, let's run the program:

```
go run hello.go
```

You'll receive the following output:

Output

```
Hello, World!
```

Notice that we also introduced a function called `main()`. The `main()` function is a special function that tells the compiler that this is where the program should start. For any program that you want to be executable (a program that can be run from the command line), you will need a `main()` function. The `main()` function must appear only once, be in the `main()` package, and receive and return no arguments. This allows for program execution in any Go program. As per the following example:

main.go

```
package main

import "fmt"

func main() {
    fmt.Println("this is the main section of the program")
}
```

Functions can be more complicated than the `hello()` function we defined. We can use for loops, conditional statements, and more within our function block.

For example, the following function uses a conditional statement to check if the input for the `name` variable contains a vowel, then uses a `for` loop to iterate over the letters in

the name string.

names.go

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    names()
}

func names() {
    fmt.Println("Enter your name:")

    var name string
    fmt.Scanln(&name)
    // Check whether name has a vowel
    for _, v := range strings.ToLower(name) {
        if v == 'a' || v == 'e' || v == 'i' || v == 'o' || v == 'u' {
            fmt.Println("Your name contains a vowel.")
            return
        }
    }
    fmt.Println("Your name does not contain a vowel.")
}
```

The `names()` function we define here sets up a `name` variable with input, and then sets up a conditional statement within a `for` loop. This shows how code can be organized within a function definition. However, depending on what we intend with our program and how we want to set up our code, we may want to define the conditional statement and the `for` loop as two separate functions.

Defining functions within a program makes our code modular and reusable so that we can call the same functions without rewriting them.

Working with Parameters

So far we have looked at functions with empty parentheses that do not take arguments, but we can define parameters in function definitions within their parentheses.

A parameter is a named entity in a function definition, specifying an argument that the function can accept. In Go, you must specify the data type for each parameter.

Let's create a program that repeats a word a specified number of times. It will take a `string` parameter called `word` and an `int` parameter called `reps` for the number of times to repeat the word.

repeat.go

```
package main

import "fmt"

func main() {
    repeat("Sammy", 5)
}

func repeat(word string, reps int) {
    for i := 0; i < reps; i++ {
        fmt.Print(word)
    }
}
```

We passed the value `Sammy` in for the `word` parameter, and `5` for the `reps` parameter. These values correspond with each parameter in the order they were given. The `repeat` function has a `for` loop that will iterate the number of times specified by the `reps` parameter. For each iteration, the value of the `word` parameter is printed.

Here is the output of the program:

Output

```
SammySammySammySammySammy
```

If you have a set of parameters that are all the same value, you can omit specifying the type each time. Let's create a small program that takes in parameters `x`, `y`, and `z` that are all `int` values. We'll create a function that adds the parameters together in different configurations. The sums of these will be printed by the function. Then we'll call the function and pass numbers into the function.

add_numbers.go

```
package main

import "fmt"

func main() {
    addNumbers(1, 2, 3)
}

func addNumbers(x, y, z int) {
    a := x + y
    b := x + z
    c := y + z
    fmt.Println(a, b, c)
}
```

When we created the function signature for `addNumbers`, we did not need to specify the type each time, but only at the end.

We passed the number `1` in for the `x` parameter, `2` in for the `y` parameter, and `3` in for the `z` parameter. These values correspond with each parameter in the order they are given.

The program is doing the following math based on the values we passed to the parameters:

```
a = 1 + 2
b = 1 + 3
c = 2 + 3
```

The function also prints `a`, `b`, and `c`, and based on this math we would expect `a` to be equal to `3`, `b` to be `4`, and `c` to be `5`. Let's run the program:

```
go run add_numbers.go
```

Output

```
3 4 5
```

When we pass `1`, `2`, and `3` as parameters to the `addNumbers()` function, we receive the expected output.

Parameters are arguments that are typically defined as variables within function definitions. They can be assigned values when you run the method, passing the arguments into the function.

Returning a Value

You can pass a parameter value into a function, and a function can also produce a value.

A function can produce a value with the `return` statement, which will exit a function and optionally pass an expression back to the caller. The return data type must be specified as well.

So far, we have used the `fmt.Println()` statement instead of the `return` statement in our functions. Let's create a program that instead of printing will return a variable.

In a new text file called `double.go`, we'll create a program that doubles the parameter `x` and returns the variable `y`. We issue a call to print the `result` variable, which is formed by running the `double()` function with `3` passed into it:

double.go

```
package main

import "fmt"

func main() {
    result := double(3)
    fmt.Println(result)
}

func double(x int) int {
    y := x * 2
    return y
}
```

We can run the program and see the output:

```
go run double.go
```

Output

```
6
```

The integer `6` is returned as output, which is what we would expect by multiplying `3` by `2`.

If a function specifies a return, you must provide a return as part of the code. If you do not, you will receive a compilation error.

We can demonstrate this by commenting out the line with the return statement:

double.go

```go
package main

import "fmt"

func main() {
    result := double(3)
    fmt.Println(result)
}

func double(x int) int {
    y := x * 2
    // return y
}
```

Now, let's run the program again:

```
go run double.go
```

Output

```
./double.go:13:1: missing return at end of function
```

Without using the `return` statement here, the program cannot compile.

Functions exit immediately when they hit a `return` statement, even if they are not at the end of the function:

return_loop.go

```
package main

import "fmt"

func main() {
    loopFive()
}

func loopFive() {
    for i := 0; i < 25; i++ {
        fmt.Print(i)
        if i == 5 {
            // Stop function at i == 5
            return
        }
    }
    fmt.Println("This line will not execute.")
}
```

Here we iterate through a `for` loop, and tell the loop to run `25` iterations. However, inside the `for` loop, we have a conditional `if` statement that checks to see if the value of `i` is equal to `5`. If it is, we issue a `return` statement. Because we are in the `loopFive` function, any `return` at any point in the function will exit the function. As a result, we never get to the last line in this function to print the statement `This line will not execute.`.

Using the `return` statement within the `for` loop ends the function, so the line that is outside of the loop will not run. If, instead, we had used a [break statement](), only the loop would have exited at that time, and the last `fmt.Println()` line would run.

The `return` statement exits a function, and may return a value if specified in the function signature.

Returning Multiple Values

More than one return value can be specified for a function. Let's examine the `repeat.go` program and make it return two values. The first will be the repeated value and the second will be an error if the `reps` parameter is not a value greater than `0`:

repeat.go

```
package main

import "fmt"

func main() {
    val, err := repeat("Sammy", -1)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(val)
}

func repeat(word string, reps int) (string, error) {
    if reps <= 0 {
        return "", fmt.Errorf("invalid value of %d provided for reps. value mu
    }
    var value string
    for i := 0; i < reps; i++ {
        value = value + word
    }
    return value, nil
}
```

The first thing the `repeat` function does is check to see if the `reps` argument is a valid value. Any value that is not greater than `0` will cause an error. Since we passed in the value of `-1`, this branch of code will execute. Notice that when we return from the function, we have to provide both the `string` and `error` return values. Because the provided arguments resulted in an error, we will pass back a blank string for the first return value, and the error for the second return value.

In the `main()` function, we can receive both return values by declaring two new variables, `value` and `err`. Because there could be an error in the return, we want to check to see if we received an error before continuing on with our program. In this example, we did receive an error. We print out the error and `return` out of the `main()` function to exit the program.

If there was not an error, we would print out the return value of the function.

Note: It is considered best practice to only return two or three values. Additionally, you should always return all errors as the last return value from a function.

Running the program will result in the following output:

Output

```
invalid value of -1 provided for reps. value must be greater than 0.
```

In this section we reviewed how we can use the `return` statement to return multiple values from a function.

Conclusion

Functions are code blocks of instructions that perform actions within a program, helping to make our code reusable and modular.

To learn more about how to make your code more modular, you can read our guide on How To Write Packages in Go.

How To Use Variadic Functions in Go

Written by Gopher Guides

A variadic function is a function that accepts zero, one, or more values as a single argument. While variadic functions are not the common case, they can be used to make your code cleaner and more readable.

Variadic functions are more common than they seem. The most common one is the `Println` function from the `fmt` package.

```
func Println(a ...interface{}) (n int, err error)
```

A function with a parameter that is preceded with a set of ellipses (`...`) is considered a variadic function. The ellipsis means that the parameter provided can be zero, one, or more values. For the `fmt.Println` package, it is stating that the parameter `a` is variadic.

Let's create a program that uses the `fmt.Println` function and pass in zero, one, or more values:

print.go

```
package main

import "fmt"

func main() {
    fmt.Println()
    fmt.Println("one")
    fmt.Println("one", "two")
    fmt.Println("one", "two", "three")
}
```

The first time we call `fmt.Println`, we don't pass any arguments. The second time we call `fmt.Println` we pass in only a single argument, with the value of `one`. Then we pass `one` and `two`, and finally `one`, `two`, and `three`.

Let's run the program with the following command:

```
go run print.go
```

We'll see the following output:

Output

```
one
one two
one two three
```

The first line of the output is blank. This is because we didn't pass any arguments the first time `fmt.Println` was called. The second time the value of `one` was printed. Then `one` and `two`, and finally `one`, `two`, and `three`.

Now that we have seen how to call a variadic function, let's look at how we can define our own variadic function.

Defining a Variadic Function

We can define a variadic function by using an ellipsis (`...`) in front of the argument. Let's create a program that greets people when their names are sent to the function:

hello.go

```go
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```

We created a `sayHello` function that takes only a single parameter called `names`. The parameter is variadic, as we put an ellipsis (`...`) before the data type:

`...string`. This tells Go that the function can accept zero, one, or many arguments.

The `sayHello` function receives the `names` parameter as a [`slice`](). Since the data type is a [`string`](), the `names` parameter can be treated just like a slice of strings (`[]string`) inside the function body. We can create a loop with the [`range`]() operator and iterate through the slice of strings.

If we run the program, we will get the following output:

Output

```
Hello Sammy
Hello Sammy
Hello Jessica
Hello Drew
Hello Jamie
```

Notice that nothing printed for the first time we called `sayHello`. This is because the variadic parameter was an empty `slice` of `string`. Since we are looping through the slice, there is nothing to iterate through, and `fmt.Printf` is never called.

Let's modify the program to detect that no values were sent in:

hello.go

```go
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    if len(names) == 0 {
        fmt.Println("nobody to greet")
        return
    }
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```

Now, by using an [if statement](), if no values are passed, the length of `names` will be `0`, and we will print out `nobody to greet`:

Output

```
nobody to greet
Hello Sammy
Hello Sammy
Hello Jessica
Hello Drew
Hello Jamie
```

Using a variadic parameter can make your code more readable. Let's create a function that joins words together with a specified delimiter. We'll create this program without a variadic function first to show how it would read:

join.go

```go
package main

import "fmt"

func main() {
    var line string

    line = join(",", []string{"Sammy", "Jessica", "Drew", "Jamie"})
    fmt.Println(line)

    line = join(",", []string{"Sammy", "Jessica"})
    fmt.Println(line)

    line = join(",", []string{"Sammy"})
    fmt.Println(line)
}

func join(del string, values []string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

In this program, we are passing a comma (`,`) as the delimiter to the `join` function. Then we are passing a slice of values to join. Here is the output:

Output

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

Because the function takes a slice of string as the `values` parameter, we had to wrap all of our words in a slice when we called the `join` function. This can make the code difficult to read.

Now, let's write the same function, but we'll use a variadic function:

join.go

```go
package main

import "fmt"

func main() {
    var line string

    line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
    fmt.Println(line)

    line = join(",", "Sammy", "Jessica")
    fmt.Println(line)

    line = join(",", "Sammy")
    fmt.Println(line)
}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

If we run the program, we can see that we get the same output as the previous program:

Output

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

While both versions of the `join` function do the exact same thing programmatically, the variadic version of the function is much easier to read when it is being called.

Variadic Argument Order

You can only have one variadic parameter in a function, and it must be the last parameter defined in the function. Defining parameters in a variadic function in any order other than the last parameter will result in a compilation error:

join.go

```
package main

import "fmt"

func main() {
    var line string

    line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
    fmt.Println(line)

    line = join(",", "Sammy", "Jessica")
    fmt.Println(line)

    line = join(",", "Sammy")
    fmt.Println(line)
}

func join(values ...string, del string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

This time we put the `values` parameter first in the `join` function. This will cause the following compilation error:

Output

```
./join_error.go:18:11: syntax error: cannot use ... with non-final
parameter values
```

When defining any variadic function, only the last parameter can be variadic.

Exploding Arguments

So far, we have seen that we can pass zero, one, or more values to a variadic function. However, there will be occasions when we have a slice of values and we want to send them to a variadic function.

Let's look at our `join` function from the last section to see what happens:

join.go

```
package main

import "fmt"

func main() {
    var line string

    names := []string{"Sammy", "Jessica", "Drew", "Jamie"}

    line = join(",", names)
    fmt.Println(line)
}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

If we run this program, we will receive a compilation error:

Output

```
./join-error.go:10:14: cannot use names (type []string) as type
string in argument to join
```

Even though the variadic function will convert the parameter of `values ...string` to a slice of strings `[]string`, we can't pass a slice of strings as the argument. This is because the compiler expects discrete arguments of strings.

To work around this, we can explode a slice by suffixing it with a set of ellipses (`...`) and turning it into discrete arguments that will be passed to a variadic function:

join.go

```
package main

import "fmt"

func main() {
    var line string

    names := []string{"Sammy", "Jessica", "Drew", "Jamie"}

    line = join(",", names...)
    fmt.Println(line)
}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

This time, when we called the `join` function, we exploded the `names` slice by appending ellipses (`...`).

This allows the program to now run as expected:

Output

```
Sammy,Jessica,Drew,Jamie
```

It's important to note that we can still pass a zero, one, or more arguments, as well as a slice that we explode. Here is the code passing all the variations that we have seen so far:

join.go

```go
package main

import "fmt"

func main() {
    var line string

    line = join(",", []string{"Sammy", "Jessica", "Drew", "Jamie"}...)
    fmt.Println(line)

    line = join(",", "Sammy", "Jessica", "Drew", "Jamie")
    fmt.Println(line)

    line = join(",", "Sammy", "Jessica")
    fmt.Println(line)

    line = join(",", "Sammy")
    fmt.Println(line)

}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

Output

```
Sammy,Jessica,Drew,Jamie
Sammy,Jessica,Drew,Jamie
Sammy,Jessica
Sammy
```

We now know how to pass zero, one, or many arguments, as well as a slice that we explode, to a variadic function.

Conclusion

In this tutorial, we have seen how variadic functions can make your code cleaner. While you won't always need to use them, you may find them useful:

- If you find that you're creating a temporary slice just to pass to a function.
- When the number of input params are unknown or will vary when called.
- To make your code more readable.

To learn more about creating and calling functions, you can read How to Define and Call Functions in Go.

Understanding defer in Go

Written by Gopher Guides

Go has many of the common control-flow keywords found in other programming languages such as `if`, `switch`, `for`, etc. One keyword that isn't found in most other programming languages is `defer`, and though it's less common you'll quickly see how useful it can be in your programs.

One of the primary uses of a `defer` statement is for cleaning up resources, such as open files, network connections, and database handles. When your program is finished with these resources, it's important to close them to avoid exhausting the program's limits and to allow other programs access to those resources. `defer` makes our code cleaner and less error prone by keeping the calls to close the file/resource in proximity to the open call.

In this article we will learn how to properly use the `defer` statement for cleaning up resources as well as several common mistakes that are made when using `defer`.

What is a `defer` Statement

A `defer` statement adds the function call following the `defer` keyword onto a stack. All of the calls on that stack are called when the function in which they were added returns. Because the calls are placed on a stack, they are called in last-in-first-out order.

Let's look at how `defer` works by printing out some text:

main.go

```
package main

import "fmt"

func main() {
    defer fmt.Println("Bye")
    fmt.Println("Hi")
}
```

In the `main` function, we have two statements. The first statement starts with the `defer` keyword, followed by a `print` statement that prints out `Bye`. The next line prints out `Hi`.

If we run the program, we will see the following output:

Output

```
Hi
Bye
```

Notice that `Hi` was printed first. This is because any statement that is preceded by the `defer` keyword isn't invoked until the end of the function in which `defer` was used.

Let's take another look at the program, and this time we'll add some comments to help illustrate what is happening:

main.go

```
package main

import "fmt"

func main() {
    // defer statement is executed, and places
    // fmt.Println("Bye") on a list to be executed prior to the function retu
    defer fmt.Println("Bye")

    // The next line is executed immediately
    fmt.Println("Hi")

    // fmt.Println*("Bye") is now invoked, as we are at the end of the functi
}
```

The key to understanding `defer` is that when the `defer` statement is executed, the arguments to the deferred function are evaluated immediately. When a `defer` executes, it places the statement following it on a list to be invoked prior to the function returning.

Although this code illustrates the order in which `defer` would be run, it's not a typical way it would be used when writing a Go program. It's more likely we are using `defer` to clean up a resource, such as a file handle. Let's look at how to do that next.

Using `defer` to Clean Up Resources

Using `defer` to clean up resources is very common in Go. Let's first look at a program that writes a string to a file but does not use `defer` to handle the resource clean-up:

main.go

```go
package main

import (
    "io"
    "log"
    "os"
)

func main() {
    if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
    }
}

func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
        return err
    }
    _, err = io.WriteString(file, text)
    if err != nil {
        return err
    }
    file.Close()
    return nil
}
```

In this program, there is a function called `write` that will first attempt to create a file. If it has an error, it will return the error and exit the function. Next, it tries to write the string `This is a readme file` to the specified file. If it receives an error, it will return the error and exit the function. Then, the function will try to close the file and release the resource back to the system. Finally the function returns `nil` to signify that the function executed without error.

Although this code works, there is a subtle bug. If the call to `io.WriteString` fails, the function will return without closing the file and releasing the resource back to the system.

We could fix the problem by adding another `file.Close()` statement, which is how you would likely solve this in a language without `defer`:

main.go

```
package main

import (
    "io"
    "log"
    "os"
)

func main() {
    if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
    }
}

func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
        return err
    }
    _, err = io.WriteString(file, text)
    if err != nil {
        file.Close()
        return err
    }
    file.Close()
    return nil
}
```

Now even if the call to `io.WriteString` fails, we will still close the file. While this was a relatively easy bug to spot and fix, with a more complicated function, it may have been missed.

Instead of adding the second call to `file.Close()`, we can use a `defer` statement to ensure that regardless of which branches are taken during execution, we always call `Close()`.

Here's the version that uses the `defer` keyword:

main.go

```go
package main

import (
    "io"
    "log"
    "os"
)

func main() {
    if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
    }
}

func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
        return err
    }
    defer file.Close()
    _, err = io.WriteString(file, text)
    if err != nil {
        return err
    }
    return nil
}
```

This time we added the line of code: `defer file.Close()`. This tells the compiler that it should execute the `file.Close` prior to exiting the function `write`.

We have now ensured that even if we add more code and create another branch that exits the function in the future, we will always clean up and close the file.

However, we have introduced yet another bug by adding the defer. We are no longer checking the potential error that can be returned from the `Close` method. This is because when we use `defer`, there is no way to communicate any return value back to our function.

In Go, it is considered a safe and accepted practice to call `Close()` more than once without affecting the behavior of your program. If `Close()` is going to return an error, it will do so the first time it is called. This allows us to call it explicitly in the successful path of execution in our function.

Let’s look at how we can both `defer` the call to `Close`, and still report on an error if we encounter one.

main.go

```
package main

import (
    "io"
    "log"
    "os"
)

func main() {
    if err := write("readme.txt", "This is a readme file"); err != nil {
        log.Fatal("failed to write file:", err)
    }
}

func write(fileName string, text string) error {
    file, err := os.Create(fileName)
    if err != nil {
        return err
    }
    defer file.Close()
    _, err = io.WriteString(file, text)
    if err != nil {
        return err
    }

    return file.Close()
}
```

The only change in this program is the last line in which we return `file.Close()`. If the call to `Close` results in an error, this will now be returned as expected to the calling function. Keep in mind that our `defer file.Close()` statement is also going to run after the `return` statement. This means that `file.Close()` is potentially called twice. While this isn’t ideal, it is an acceptable practice as it should not create any side effects to your program.

If, however, we receive an error earlier in the function, such as when we call `WriteString`, the function will return that error, and will also try to call `file.Close`

because it was deferred. Although `file.Close` may (and likely will) return an error as well, it is no longer something we care about as we received an error that is more likely to tell us what went wrong to begin with.

So far, we have seen how we can use a single `defer` to ensure that we clean up our resources properly. Next we will see how we can use multiple `defer` statements for cleaning up more than one resource.

Multiple `defer` Statements

It is normal to have more than one `defer` statement in a function. Let's create a program that only has `defer` statements in it to see what happens when we introduce multiple defers:

main.go

```go
package main

import "fmt"

func main() {
    defer fmt.Println("one")
    defer fmt.Println("two")
    defer fmt.Println("three")
}
```

If we run the program, we will receive the following output:

Output

```
three
two
one
```

Notice that the order is the opposite in which we called the `defer` statements. This is because each deferred statement that is called is stacked on top of the previous one, and then called in reverse when the function exits scope (Last In, First Out).

You can have as many deferred calls as needed in a function, but it is important to remember they will all be called in the opposite order they were executed.

Now that we understand the order in which multiple defers will execute, let's see how we would use multiple defers to clean up multiple resources. We'll create a program that opens a file, writes to it, then opens it again to copy the contents to another file.

main.go

```go
package main

import (
	"fmt"
	"io"
	"log"
	"os"
)

func main() {
	if err := write("sample.txt", "This file contains some sample text."); er
		log.Fatal("failed to create file")
	}

	if err := fileCopy("sample.txt", "sample-copy.txt"); err != nil {
		log.Fatal("failed to copy file: %s")
	}
}

func write(fileName string, text string) error {
	file, err := os.Create(fileName)
	if err != nil {
		return err
	}
	defer file.Close()
	_, err = io.WriteString(file, text)
	if err != nil {
		return err
	}

	return file.Close()
}

func fileCopy(source string, destination string) error {
	src, err := os.Open(source)
	if err != nil {
		return err
	}
	defer src.Close()
```

```
    dst, err := os.Create(destination)
    if err != nil {
        return err
    }
    defer dst.Close()

    n, err := io.Copy(dst, src)
    if err != nil {
        return err
    }
    fmt.Printf("Copied %d bytes from %s to %s\n", n, source, destination)

    if err := src.Close(); err != nil {
        return err
    }

    return dst.Close()
}
```

We added a new function called `fileCopy`. In this function, we first open up our source file that we are going to copy from. We check to see if we received an error opening the file. If so, we `return` the error and exit the function. Otherwise, we `defer` the closing of the source file we just opened.

Next we create the destination file. Again, we check to see if we received an error creating the file. If so, we `return` that error and exit the function. Otherwise, we also `defer` the `Close()` for the destination file. We now have two `defer` functions that will be called when the function exits its scope.

Now that we have both files open, we will `Copy()` the data from the source file to the destination file. If that is successful, we will attempt to close both files. If we receive an error trying to close either file, we will `return` the error and exit function scope.

Notice that we explicitly call `Close()` for each file, even though the `defer` will also call `Close()`. This is to ensure that if there is an error closing a file, we report the error. It also ensures that if for any reason the function exits early with an error, for instance if we failed to copy between the two files, that each file will still try to close properly from the deferred calls.

Conclusion

In this article we learned about the `defer` statement, and how it can be used to ensure that we properly clean up system resources in our program. Properly cleaning up system

resources will make your program use less memory and perform better. To learn more about where `defer` is used, read the article on Handling Panics, or explore our entire How To Code in Go series.

Understanding init in Go

Written by Gopher Guides

In Go, the predefined `init()` function sets off a piece of code to run before any other part of your package. This code will execute as soon as the package is imported, and can be used when you need your application to initialize in a specific state, such as when you have a specific configuration or set of resources with which your application needs to start. It is also used when importing a side effect, a technique used to set the state of a program by importing a specific package. This is often used to `register` one package with another to make sure that the program is considering the correct code for the task.

Although `init()` is a useful tool, it can sometimes make code difficult to read, since a hard-to-find `init()` instance will greatly affect the order in which the code is run. Because of this, it is important for developers who are new to Go to understand the facets of this function, so that they can make sure to use `init()` in a legible manner when writing code.

In this tutorial, you'll learn how `init()` is used for the setup and initialization of specific package variables, one time computations, and the registration of a package for use with another package.

Prerequisites

For some of the examples in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment. This tutorial will use the following file structure:

```
.
├── bin
│
└── src
    └── github.com
        └── gopherguides
```

Declaring `init()`

Any time you declare an `init()` function, Go will load and run it prior to anything else in that package. To demonstrate this, this section will walk through how to define an `init()` function and show the effects on how the package runs.

Let's first take the following as an example of code without the `init()` function:

main.go

```
package main

import "fmt"

var weekday string

func main() {
    fmt.Printf("Today is %s", weekday)
}
```

In this program, we declared a global variable called `weekday`. By default, the value of `weekday` is an empty string.

Let's run this code:

```
go run main.go
```

Because the value of `weekday` is blank, when we run the program, we will get the following output:

Output

```
Today is
```

We can fill in the blank variable by introducing an `init()` function that initializes the value of `weekday` to the current day. Add in the following highlighted lines to `main.go`:

main.go

```
package main

import (
    "fmt"
    "time"
)

var weekday string

func init() {
    weekday = time.Now().Weekday().String()
}

func main() {
    fmt.Printf("Today is %s", weekday)
}
```

In this code, we imported and used the `time` package to get the current day of the week (`Now().Weekday().String()`), then used `init()` to initialize `weekday` with that value.

Now when we run the program, it will print out the current weekday:

Output

```
Today is Monday
```

While this illustrates how `init()` works, a much more typical use case for `init()` is to use it when importing a package. This can be useful when you need to do specific setup tasks in a package before you want the package to be used. To demonstrate this, let's create a program that will require a specific initialization for the package to work as intended.

Initializing Packages on Import

First, we will write some code that selects a random creature from a slice and prints it out. However, we won't use `init()` in our initial program. This will better show the problem we have, and how `init()` will solve our problem.

From within your `src/github.com/gopherguides/` directory, create a folder called `creature` with the following command:

```
mkdir creature
```

Inside the `creature` folder, create a file called `creature.go`:

```
nano creature/creature.go
```

In this file, add the following contents:

creature.go

```
package creature

import (
    "math/rand"
)

var creatures = []string{"shark", "jellyfish", "squid", "octopus", "dolphin"}

func Random() string {
    i := rand.Intn(len(creatures))
    return creatures[i]
}
```

This file defines a variable called `creatures` that has a set of sea creatures initialized as values. It also has an exported `Random` function that will return a random value from the `creatures` variable.

Save and quit this file.

Next, let's create a `cmd` package that we will use to write our `main()` function and call the `creature` package.

At the same file level from which we created the `creature` folder, create a `cmd` folder with the following command:

```
mkdir cmd
```

Inside the `cmd` folder, create a file called `main.go`:

```
nano cmd/main.go
```

Add the following contents to the file:

cmd/main.go

```go
package main

import (
    "fmt"

    "github.com/gopherguides/creature"
)

func main() {
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
    fmt.Println(creature.Random())
}
```

Here we imported the `creature` package, and then in the `main()` function, used the `creature.Random()` function to retrieve a random creature and print it out four times.

Save and quit `main.go`.

We now have our entire program written. However, before we can run this program, we'll need to also create a couple of configuration files for our code to work properly. Go uses [Go Modules](https://blog.golang.org/using-go-modules) to configure package dependencies for importing resources. These modules are configuration files placed in your package directory that tell the compiler where to import packages from. While learning about modules is beyond the scope of this article, we can write just a couple lines of configuration to make this example work locally.

In the `cmd` directory, create a file named `go.mod`:

```
nano cmd/go.mod
```

Once the file is open, place in the following contents:

cmd/go.mod

```
module github.com/gopherguides/cmd
 replace github.com/gopherguides/creature => ../creature
```

The first line of this file tells the compiler that the `cmd` package we created is in fact `github.com/gopherguides/cmd`. The second line tells the compiler that `github.com/gopherguides/creature` can be found locally on disk in the `../creature` directory.

Save and close the file. Next, create a `go.mod` file in the `creature` directory:

```
nano creature/go.mod
```

Add the following line of code to the file:

creature/go.mod

```
module github.com/gopherguides/creature
```

This tells the compiler that the `creature` package we created is actually the `github.com/gopherguides/creature` package. Without this, the `cmd` package would not know where to import this package from.

Save and quit the file.

You should now have the following directory structure and file layout:

```
├── cmd
│   ├── go.mod
│   └── main.go
└── creature
    ├── go.mod
    └── creature.go
```

Now that we have all the configuration completed, we can run the `main` program with the following command:

```
go run cmd/main.go
```

This will give:

Output

```
jellyfish
squid
squid
dolphin
```

When we ran this program, we received four values and printed them out. If we run the program multiple times, we will notice that we always get the same output, rather than a random result as expected. This is because the `rand` package creates pseudorandom numbers that will consistently generate the same output for a single initial state. To achieve a more random number, we can seed the package, or set a changing source so that the initial state is different every time we run the program. In Go, it is common to use the current time to seed the `rand` package.

Since we want the `creature` package to handle the random functionality, open up this file:

```
nano creature/creature.go
```

Add the following highlighted lines to the `creature.go` file:

creature/creature.go

```
package creature

import (
    "math/rand"
    "time"
)

var creatures = []string{"shark", "jellyfish", "squid", "octopus",
"dolphin"}

func Random() string {
    rand.Seed(time.Now().UnixNano())
    i := rand.Intn(len(creatures))
    return creatures[i]
}
```

In this code, we imported the `time` package and used `Seed()` to seed the current time. Save and exit the file.

Now, when we run the program we will get a random result:

```
go run cmd/main.go
```

Output

```
jellyfish
octopus
shark
jellyfish
```

If you continue to run the program over and over, you will continue to get random results. However, this is not yet an ideal implementation of our code, because every time `creature.Random()` is called, it also re-seeds the `rand` package by calling `rand.Seed(time.Now().UnixNano())` again. Re-seeding will increase the chance of seeding with the same initial value if the internal clock has not changed, which will cause possible repetitions of the random pattern, or will increase CPU processing time by having your program wait for the clock to change.

To fix this, we can use an `init()` function. Let's update the `creature.go` file:

```
nano creature/creature.go
```

Add the following lines of code:

creature/creature.go

```
package creature

import (
    "math/rand"
    "time"
)

var creatures = []string{"shark", "jellyfish", "squid", "octopus",
"dolphin"}

func init() {
    rand.Seed(time.Now().UnixNano())
}

func Random() string {
    i := rand.Intn(len(creatures))
    return creatures[i]
}
```

Adding the `init()` function tells the compiler that when the `creature` package is imported, it should run the `init()` function once, providing a single seed for the random number generation. This ensures that we don't run code more than we have to. Now if we run the program, we will continue to get random results:

```
go run cmd/main.go
```

Output

```
dolphin
squid
dolphin
octopus
```

In this section, we have seen how using `init()` can ensure that the appropriate calculations or initializations are performed prior to the package being used. Next, we will see how to use multiple `init()` statements in a package.

Multiple Instances of `init()`

Unlike the `main()` function that can only be declared once, the `init()` function can be declared multiple times throughout a package. However, multiple `init()`s can make it

difficult to know which one has priority over the others. In this section, we will show how to maintain control over multiple `init()` statements.

In most cases, `init()` functions will execute in the order that you encounter them. Let's take the following code as an example:

main.go

```
package main

import "fmt"

func init() {
    fmt.Println("First init")
}

func init() {
    fmt.Println("Second init")
}

func init() {
    fmt.Println("Third init")
}

func init() {
    fmt.Println("Fourth init")
}

func main() {}
```

If we run the program with the following command:

```
go run main.go
```

We'll receive the following output:

Output

```
First init
Second init
Third init
Fourth init
```

Notice that each `init()` is run in the order in which the compiler encounters it. However, it may not always be so easy to determine the order in which the `init()`

function will be called.

Let's look at a more complicated package structure in which we have multiple files each with their own `init()` function declared in them. To illustrate this, we will create a program that shares a variable called `message` and prints it out.

Delete the `creature` and `cmd` directories and their contents from the earlier section, and replace them with the following directories and file structure:

```
├── cmd
│   ├── a.go
│   ├── b.go
│   └── main.go
└── message
    └── message.go
```

Now let's add the contents of each file. In `a.go`, add the following lines:

cmd/a.go

```
package main

import (
    "fmt"

    "github.com/gopherguides/message"
)

func init() {
    fmt.Println("a ->", message.Message)
}
```

This file contains a single `init()` function that prints out the value of `message.Message` from the `message` package.

Next, add the following contents to `b.go`:

cmd/b.go

```go
package main

import (
	"fmt"

	"github.com/gopherguides/message"
)

func init() {
	message.Message = "Hello"
	fmt.Println("b ->", message.Message)
}
```

In `b.go`, we have a single `init()` function that set's the value of `message.Message` to `Hello` and prints it out.

Next, create `main.go` to look like the following:

cmd/main.go

```go
package main

func main() {}
```

This file does nothing, but provides an entry point for the program to run.

Finally, create your `message.go` file like the following:

message/message.go

```go
package message

var Message string
```

Our `message` package declares the exported `Message` variable.

To run the program, execute the following command from the `cmd` directory:

```
go run *.go
```

Because we have multiple Go files in the `cmd` folder that make up the `main` package, we need to tell the compiler that all the `.go` files in the `cmd` folder should be compiled. Using `*.go` tells the compiler to load all the files in the `cmd` folder that end in `.go`. If we issued the command of `go run main.go`, the program would fail to compile as it would not see the code in the `a.go` and `b.go` files.

This will give the following output:

Output

```
a ->
b -> Hello
```

Per the Go language specification for Package Initialization, when multiple files are encountered in a package, they are processed alphabetically. Because of this, the first time we printed out `message.Message` from `a.go`, the value was blank. The value wasn't initialized until the `init()` function from `b.go` had been run.

If we were to change the file name of `a.go` to `c.go`, we would get a different result:

Output

```
b -> Hello
a -> Hello
```

Now the compiler encounters `b.go` first, and as such, the value of `message.Message` is already initialized with `Hello` when the `init()` function in `c.go` is encountered.

This behavior could create a possible problem in your code. It is common in software development to change file names, and because of how `init()` is processed, changing file names may change the order in which `init()` is processed. This could have the undesired effect of changing your program's output. To ensure reproducible initialization behavior, build systems are encouraged to present multiple files belonging to the same package in lexical file name order to a compiler. One way to ensure that all `init()` functions are loaded in order is to declare them all in one file. This will prevent the order from changing even if file names are changed.

In addition to ensuring the order of your `init()` functions does not change, you should also try to avoid managing state in your package by using global variables, i.e., variables that are accessible from anywhere in the package. In the preceding program, the `message.Message` variable was available to the entire package and maintained the state of the program. Because of this access, the `init()` statements were able to change the variable and destablize the predictability of your program. To avoid this, try to work with variables in controlled spaces that have as little access as possible while still allowing the program to work.

We have seen that you can have multiple `init()` declarations in a single package. However, doing so may create undesired effects and make your program hard to read or predict. Avoiding multiple `init()` statements or keeping them all in one file will ensure that the behavior of your program does not change when files are moved around or names are changed.

Next, we will examine how `init()` is used to import with side effects.

Using `init()` for Side Effects

In Go, it is sometimes desirable to import a package not for its content, but for the side effects that occur upon importing the package. This often means that there is an `init()` statement in the imported code that executes before any of the other code, allowing for the developer to manipulate the state in which their program is starting. This technique is called importing for a side effect.

A common use case for importing for side effects is to register functionality in your code, which lets a package know what part of the code your program needs to use. In the [image package](), for example, the `image.Decode` function needs to know which format of image it is trying to decode (`jpg`, `png`, `gif`, etc.) before it can execute. You can accomplish this by first importing a specific program that has an `init()` statement side effect.

Let's say you are trying to use `image.Decode` on a `.png` file with the following code snippet:

Sample Decoding Snippet

```
. . .
func decode(reader io.Reader) image.Rectangle {
    m, _, err := image.Decode(reader)
    if err != nil {
        log.Fatal(err)
    }
    return m.Bounds()
}
. . .
```

A program with this code will still compile, but any time we try to decode a `png` image, we will get an error.

To fix this, we would need to first register an image format for `image.Decode`. Luckily, the `image/png` package contains the following `init()` statement:

image/png/reader.go

```
func init() {
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

Therefore, if we import `image/png` into our decoding snippet, then the `image.RegisterFormat()` function in `image/png` will run before any of our code:

Sample Decoding Snippet

```
. . .
import _ "image/png"
. . .

func decode(reader io.Reader) image.Rectangle {
    m, _, err := image.Decode(reader)
    if err != nil {
        log.Fatal(err)
    }
    return m.Bounds()
}
```

This will set the state and register that we require the `png` version of `image.Decode()`. This registration will happen as a side effect of importing `image/png`.

You may have noticed the blank identifier (_) before `"image/png"`. This is needed because Go does not allow you to import packages that are not used throughout the program. By including the blank identifier, the value of the import itself is discarded, so that only the side effect of the import comes through. This means that, even though we never call the `image/png` package in our code, we can still import it for the side effect.

It is important to know when you need to import a package for its side effect. Without the proper registration, it is likely that your program will compile, but not work properly when it is run. The packages in the standard library will declare the need for this type of import in their documentation. If you write a package that requires importing for side effect, you should also make sure the `init()` statement you are using is documented so users that import your package will be able to use it properly.

Conclusion

In this tutorial we learned that the `init()` function loads before the rest of the code in your package is loaded, and that it can perform specific tasks for a package like initializing a desired state. We also learned that the order in which the compiler executes multiple `init()` statements is dependent on the order in which the compiler loads the source files. If you would like to learn more about `init()`, check out the official Golang documentation, or read through the discussion in the Go community about the function.

You can read more about functions with our How To Define and Call Functions in Go article, or explore the entire How To Code in Go series.

Customizing Go Binaries with Build Tags

Written by Gopher Guides

In Go, a build tag, or a build constraint, is an identifier added to a piece of code that determines when the file should be included in a package during the `build` process. This allows you to build different versions of your Go application from the same source code and to toggle between them in a fast and organized manner. Many developers use build tags to improve the workflow of building cross-platform compatible applications, such as programs that require code changes to account for variances between different operating systems. Build tags are also used for integration testing, allowing you to quickly switch between the integrated code and the code with a mock service or stub, and for differing levels of feature sets within an application.

Let's take the problem of differing customer feature sets as an example. When writing some applications, you may want to control which features to include in the binary, such as an application that offers Free, Pro, and Enterprise levels. As the customer increases their subscription level in these applications, more features become unlocked and available. To solve this problem, you could maintain

separate projects and try to keep them in sync with each other through the use of `import` statements. While this approach would work, over time it would become tedious and error prone. An alternative approach would be to use build tags.

In this article, you will use build tags in Go to generate different executable binaries that offer Free, Pro, and Enterprise feature sets of a sample application. Each will have a different set of features available, with the Free version being the default.

Prerequisites

To follow the example in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment.

Building the Free Version

Let’s start by building the Free version of the application, as it will be the default when running `go build` without any build tags. Later on, we will use build tags to selectively add other parts to our program.

In the `src` directory, create a folder with the name of your application. This tutorial will use `app`:

```
mkdir app
```

Move into this folder:

```
cd app
```

Next, make a new text file in your text editor of choice named `main.go`:

```
nano main.go
```

Now, we'll define the Free version of the application. Add in the following contents to `main.go`:

main.go

```
package main

import "fmt"

var features = []string{
  "Free Feature #1",
  "Free Feature #2",
}

func main() {
  for _, f := range features {
    fmt.Println(">", f)
  }
}
```

In this file, we created a program that declares a slice named `features`, which holds two strings that represent the features of our Free application. The `main()` function in the application uses a for loop to range through the

`features` slice and print all of the features available to the screen.

Save and exit the file. Now that this file is saved, we will no longer have to edit it for the rest of the article. Instead we will use build tags to change the features of the binaries we will build from it.

Build and run the program:

```
go build
./app
```

You'll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
```

The program has printed out our two free features, completing the Free version of our app.

So far, you created an application that has a very basic feature set. Next, you will build a way to add more features into the application at build time.

Adding the Pro Features With `go build`

We have so far avoided making changes to `main.go`, simulating a common production environment in which code needs to be added without changing and possibly breaking the main code. Since we can't edit the `main.go`

file, we'll need to use another mechanism for injecting more features into the `features` slice using build tags.

Let's create a new file called `pro.go` that will use an [`init()`](#) function to append more features to the `features` slice:

```
nano pro.go
```

Once the editor has opened the file, add the following lines:

pro.go

```go
package main

func init() {
  features = append(features,
    "Pro Feature #1",
    "Pro Feature #2",
  )
}
```

In this code, we used `init()` to run code before the `main()` function of our application, followed by `append()` to add the Pro features to the `features` slice. Save and exit the file.

Compile and run the application using `go build`:

```
go build
```

Since there are now two files in our current directory (`pro.go` and `main.go`), `go build` will create a binary from both of them. Execute this binary:

```
./app
```

This will give you the following feature set:

Output

```
> Free Feature #1
> Free Feature #2
> Pro Feature #1
> Pro Feature #2
```

The application now includes both the Pro and the Free features. However, this is not desirable: since there is no distinction between versions, the Free version now includes the features that are supposed to be only available in the Pro version. To fix this, you could include more code to manage the different tiers of the application, or you could use build tags to tell the Go tool chain which `.go` files to build and which to ignore. Let's add build tags in the next step.

Adding Build Tags

You can now use build tags to distinguish the Pro version of your application from the Free version.

Let's start by examining what a build tag looks like:

```
// +build tag_name
```

By putting this line of code as the first line of your package and replacing `tag_name` with the name of your build tag, you will tag this package as code that can be selectively included in the final binary. Let's see this in action by adding a build tag to the `pro.go` file to tell the `go build` command to ignore it unless the tag is specified. Open up the file in your text editor:

```
nano pro.go
```

Then add the following highlighted line:

pro.go

```
// +build pro

package main

func init() {
  features = append(features,
    "Pro Feature #1",
    "Pro Feature #2",
  )
}
```

At the top of the `pro.go` file, we added `// +build pro` followed by a blank newline. This trailing newline is required, otherwise Go interprets this as a comment. Build

tag declarations must also be at the very top of a `.go` file. Nothing, not even comments, can be above build tags.

The `+build` declaration tells the `go build` command that this isn't a comment, but instead is a build tag. The second part is the `pro` tag. By adding this tag at the top of the `pro.go` file, the `go build` command will now only include the `pro.go` file with the `pro` tag is present.

Compile and run the application again:

```
go build
./app
```

You'll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
```

Since the `pro.go` file requires a `pro` tag to be present, the file is ignored and the application compiles without it.

When running the `go build` command, we can use the `-tags` flag to conditionally include code in the compiled source by adding the tag itself as an argument. Let's do this for the `pro` tag:

```
go build -tags pro
```

This will output the following:

Output

```
> Free Feature #1
> Free Feature #2
> Pro Feature #1
> Pro Feature #2
```

Now we only get the extra features when we build the application using the `pro` build tag.

This is fine if there are only two versions, but things get complicated when you add in more tags. To add in the Enterprise version of our app in the next step, we will use multiple build tags joined together with Boolean logic.

Build Tag Boolean Logic

When there are multiple build tags in a Go package, the tags interact with each other using Boolean logic. To demonstrate this, we will add the Enterprise level of our application using both the `pro` tag and the `enterprise` tag.

In order to build an Enterprise binary, we will need to include both the default features, the Pro level features, and a new set of features for Enterprise. First, open an editor and create a new file, `enterprise.go`, that will add the new Enterprise features:

```
nano enterprise.go
```

The contents of `enterprise.go` will look almost identical to `pro.go` but will contain new features. Add the following

lines to the file:

enterprise.go

```
package main

func init() {
  features = append(features,
    "Enterprise Feature #1",
    "Enterprise Feature #2",
  )
}
```

Save and exit the file.

Currently the `enterprise.go` file does not have any build tags, and as you learned when you added `pro.go`, this means that these features will be added to the Free version when executing `go.build`. For `pro.go`, you added `// +build pro` and a newline to the top of the file to tell `go build` that it should only be included when `-tags pro` is used. In this situation, you only needed one build tag to accomplish the goal. When adding the new Enterprise features, however, you first must also have the Pro features.

Let's add support for the `pro` build tag to `enterprise.go` first. Open the file with your text editor:

```
nano enterprise.go
```

Next add the build tag before the `package main` declaration and make sure to include a newline after the build tag:

enterprise.go

```
// +build pro

package main

func init() {
  features = append(features,
    "Enterprise Feature #1",
    "Enterprise Feature #2",
  )
}
```

Save and exit the file.

Compile and run the application without any tags:

```
go build
./app
```

You'll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
```

The Enterprise features no longer show up in the Free version. Now let’s add the `pro` build tag and build and run the application again:

```
go build -tags pro
./app
```

You’ll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
> Enterprise Feature #1
> Enterprise Feature #2
> Pro Feature #1
> Pro Feature #2
```

This is still not exactly what we need: The Enterprise features now show up when we try to build the Pro version. To solve this, we need to use another build tag. Unlike the `pro` tag, however, we need to now make sure both the `pro` and `enterprise` features are available.

The Go build system accounts for this situation by allowing the use of some basic Boolean logic in the build tags system.

Let’s open `enterprise.go` again:

```
nano enterprise.go
```

Add another build tag, `enterprise`, on the same line as the `pro` tag:

enterprise.go

```
// +build pro enterprise

package main

func init() {
  features = append(features,
    "Enterprise Feature #1",
    "Enterprise Feature #2",
  )
}
```

Save and close the file.

Now let's compile and run the application with the new `enterprise` build tag.

```
go build -tags enterprise
./app
```

This will give the following:

Output

```
> Free Feature #1
> Free Feature #2
> Enterprise Feature #1
> Enterprise Feature #2
```

Now we have lost the Pro features. This is because when we put multiple build tags on the same line in a `.go` file, `go`

`build` interprets them as using `OR` logic. With the addition of the line `// +build pro enterprise`, the `enterprise.go` file will be built if either the `pro` build tag or the `enterprise` build tag is present. We need to set up the build tags correctly to require both and use `AND` logic instead.

Instead of putting both tags on the same line, if we put them on separate lines, then `go build` will interpret those tags using `AND` logic.

Open `enterprise.go` once again and let's separate the build tags onto multiple lines.

enterprise.go

```
// +build pro
// +build enterprise

package main

func init() {
  features = append(features,
    "Enterprise Feature #1",
    "Enterprise Feature #2",
  )
}
```

Now compile and run the application with the new `enterprise` build tag.

```
go build -tags enterprise
./app
```

You'll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
```

Still not quite there: Because an `AND` statement requires both elements to be considered `true`, we need to use both `pro` and `enterprise` build tags.

Let's try again:

```
go build -tags "enterprise pro"
./app
```

You'll receive the following output:

Output

```
> Free Feature #1
> Free Feature #2
> Enterprise Feature #1
> Enterprise Feature #2
> Pro Feature #1
> Pro Feature #2
```

Now our application can be built from the same source tree in multiple ways unlocking the features of the application accordingly.

In this example, we used a new `// +build` tag to signify `AND` logic, but there are alternative ways to represent Boolean logic with build tags. The following table holds some examples of other syntactic formatting for build tags, along with their Boolean equivalent:

Build Tag Syntax	Build Tag Sample	Boolean Statement
Space-separated elements	`// +build pro enterprise`	`pro` OR `enterprise`
Comma-separated elements	`// +build pro,enterprise`	`pro` AND `enterprise`
Exclamation point elements	`// +build !pro`	NOT `pro`

Conclusion

In this tutorial, you used build tags to allow you to control which of your code got compiled into the binary. First, you declared build tags and used them with `go build`, then you combined multiple tags with Boolean logic. You then built a program that represented the different feature sets of a Free, Pro, and Enterprise version, showing the powerful level of control that build tags can give you over your project.

If you'd like to learn more about build tags, take a look at the Golang documentation on the subject, or continue to

explore our How To Code in Go series.

Understanding Pointers in Go

Written by Gopher Guides

When you write software in Go you'll be writing functions and methods. You pass data to these functions as arguments. Sometimes, the function needs a local copy of the data, and you want the original to remain unchanged. For example, if you're a bank, and you have a function that shows the user the changes to their balance depending on the savings plan they choose, you don't want to change the customer's actual balance before they choose a plan; you just want to use it in calculations. This is called passing by value, because you're sending the value of the variable to the function, but not the variable itself.

Other times, you may want the function to be able to alter the data in the original variable. For instance, when the bank customer makes a deposit to their account, you want the deposit function to be able to access the actual balance, not a copy. In this case, you don't need to send the actual data to the function; you just need to tell the function where the data is located in memory. A data type called a pointer holds the memory address of the data, but not the data itself. The memory address tells the function where to find the data, but not the value of the data. You can pass the pointer to the function instead of the data, and the function can then alter the original variable in place. This is called passing by reference, because the value of the variable isn't passed to the function, just its location.

In this article, you will create and use pointers to share access to the memory space for a variable.

Defining and Using Pointers

When you use a pointer to a variable, there are a couple of different syntax elements that you need to understand. The first one is the use of the ampersand (&). If you place an ampersand in front of a variable name, you are stating that you want to get the address, or a pointer to that variable. The second syntax element is the use of the asterisk (*) or dereferencing operator. When you declare a pointer variable, you follow the variable name with the type of the variable that the pointer points to, prefixed with an *, like this:

```
var myPointer *int32 = &someint
```

This creates `myPointer` as a pointer to an `int32` variable, and initializes the pointer with the address of `someint`. The pointer doesn't actually contain an `int32`, just the address of one.

Let's take a look at a pointer to a `string`. The following code declares both a value of a string, and a pointer to a string:

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
}
```

Run the program with the following command:

```
go run main.go
```

When you run the program, it will print out the value of the variable, as well as the address of where the variable is stored (the pointer address). The memory address is a hexadecimal number, and not meant to be human-readable. In practice, you'll probably never output a memory address to look at it. We're showing you for illustrative purposes. Because each program is created in its own memory space when it is run, the value of the pointer will be different each time you run it, and will be different than the output shown here:

Output

```
creature = shark
pointer = 0xc0000721e0
```

The first variable we defined we named `creature`, and set it equal to a `string` with the value of `shark`. We then created another variable named `pointer`. This time, we set the value of the `pointer` variable to the address of the `creature` variable. We store the address of a value in a variable by using the ampersand (&) symbol. This means that the `pointer` variable is storing the address of the `creature` variable, not the actual value.

This is why when we printed out the value of `pointer`, we received the value of `0xc0000721e0`, which is the address of where the `creature` variable is currently stored in computer memory.

If you want to print out the value of the variable being pointed at from the `pointer` variable, you need to dereference that variable. The following code uses the * operator to dereference the `pointer` variable and retrieve its value:

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)

    fmt.Println("*pointer =", *pointer)
}
```

If you run this code, you'll see the following output:

Output

```
creature = shark
pointer = 0xc000010200
*pointer = shark
```

The last line we added now dereferences the `pointer` variable, and prints out the value that is stored at that address.

If you want to modify the value stored at the `pointer` variable's location, you can use the dereference operator as well:

main.go

```go
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)

    fmt.Println("*pointer =", *pointer)

    *pointer = "jellyfish"
    fmt.Println("*pointer =", *pointer)
}
```

Run this code to see the output:

Output

```
creature = shark
pointer = 0xc000094040
*pointer = shark
*pointer = jellyfish
```

We set the value the `pointer` variable refers to by using the asterisk (*) in front of the variable name, and then providing a new value of `jellyfish`. As you can see, when we print the dereferenced value, it is now set to `jellyfish`.

You may not have realized it, but we actually changed the value of the `creature` variable as well. This is because the `pointer` variable is actually pointing at the `creature` variable's address. This means that if we change the value pointed at from the `pointer` variable, we also change the value of the `creature` variable.

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)

    fmt.Println("*pointer =", *pointer)

    *pointer = "jellyfish"
    fmt.Println("*pointer =", *pointer)

    fmt.Println("creature =", creature)
}
```

The output looks like this:

Output

```
creature = shark
pointer = 0xc000010200
*pointer = shark
*pointer = jellyfish
creature = jellyfish
```

Although this code illustrates how a pointer works, this is not the typical way in which you would use pointers in Go. It is more common to use them when defining function arguments and return values, or using them when defining methods on custom types. Let's look at how you would use pointers with functions to share access to a variable.

Again, keep in mind that we are printing the value of `pointer` to illustrate that it is a pointer. In practice, you wouldn't use the value of a pointer, other than to reference the underlying value to retrieve or update that value.

Function Pointer Receivers

When you write a function, you can define arguments to be passed ether by value, or by reference. Passing by value means that a copy of that value is sent to the function, and any changes to that argument within that function only effect that variable within that function, and not where it was passed from. However, if you pass by reference, meaning you pass a pointer to that argument, you can change the value from within the function, and also change the value of the original variable that was passed in. You can read more about how to define functions in our How To Define and Call Functions in Go.

}

Deciding when to pass a pointer as opposed when to send a value is all about knowing if you want the value to change or not. If you don't want the value to change, send it as a value. If you want the function you are passing your variable to be able to change it, then you would pass it as a pointer.

To see the difference, let's first look at a function that is passing in an argument by `value`:

main.go

```
package main

import "fmt"

type Creature struct {
    Species string
}

func main() {
    var creature Creature = Creature{Species: "shark"}

    fmt.Printf("1) %+v\n", creature)
    changeCreature(creature)
    fmt.Printf("3) %+v\n", creature)
}

func changeCreature(creature Creature) {
    creature.Species = "jellyfish"
    fmt.Printf("2) %+v\n", creature)
}
```

The output looks like this:

Output

```
1) {Species:shark}
2) {Species:jellyfish}
3) {Species:shark}
```

First we created a custom type named `Creature`. It has one field named `Species`, which is a string. In the `main` function, we created an instance of our new type named `creature` and set the `Species` field to `shark`. We then printed out the variable to show the current value stored within the `creature` variable.

Next, we call `changeCreature` and pass in a copy of the `creature` variable.

The function `changeCreature` is defined as taking one argument named `creature`, and it is of type `Creature` that we defined earlier. We then change the value of the `Species` field to `jellyfish` and print it out. Notice that within the `changeCreature` function, the value of `Species` is now `jellyfish`, and it prints out `2) {Species:jellyfish}`. This is because we are allowed to change the value within our function scope.

However, when the last line of the `main` function prints the value of `creature`, the value of `Species` is still `shark`. The reason that the value didn't change is because we passed the variable by value. This means that a copy of the value was created in memory, and passed to the `changeCreature` function. This allows us to have a function that can make changes to any arguments passed in as needed, but will not affect any of those variables outside of the function.

Next, let's change the `changeCreature` function to take an argument by reference. We can do this by changing the type from `creature` to a pointer by using the asterisk (*) operator. Instead of passing a `creature`, we're now passing a pointer to a `creature`, or a `*creature`. In the previous example, `creature` is a `struct` that has a `Species` value of `shark`. `*creature` is a pointer, not a struct, so its value is a memory location, and that's what we pass to `changeCreature()`.

main.go

```
package main

import "fmt"

type Creature struct {
    Species string
}

func main() {
    var creature Creature = Creature{Species: "shark"}

    fmt.Printf("1) %+v\n", creature)
    changeCreature(&creature)
    fmt.Printf("3) %+v\n", creature)
}

func changeCreature(creature *Creature) {
    creature.Species = "jellyfish"
    fmt.Printf("2) %+v\n", creature)
}
```

Run this code to see the following output:

Output

```
1) {Species:shark}
2) &{Species:jellyfish}
3) {Species:jellyfish}
```

Notice that now when we change the value of `Species` to `jellyfish` in the `changeCreature` function, it changes the original value defined in the `main` function as well. This is because we passed the `creature` variable by reference, which allows access to the original value and can change it as needed.

Therefore, if you want a function to be able to change a value, you need to pass it by reference. To pass by reference, you pass the pointer to the variable, and not the variable itself.

However, sometimes you may not have an actual value defined for a pointer. In those cases, it is possible to have a panic in the program. Let's look at how that happens and how to plan for that potential problem.

Nil Pointers

All variables in Go have a zero value. This is true even for a pointer. If you declare a pointer to a type, but assign no value, the zero value will be `nil`. `nil` is a way to say that "nothing has been initialized" for the variable.

In the following program, we are defining a pointer to a `Creature` type, but we are never instantiating that actual instance of a `Creature` and assigning the address of it to the `creature` pointer variable. The value will be `nil` and we can't reference any of the fields or methods that would be defined on the `Creature` type:

main.go

```go
package main

import "fmt"

type Creature struct {
	Species string
}

func main() {
	var creature *Creature

	fmt.Printf("1) %+v\n", creature)
	changeCreature(creature)
	fmt.Printf("3) %+v\n", creature)
}

func changeCreature(creature *Creature) {
	creature.Species = "jellyfish"
	fmt.Printf("2) %+v\n", creature)
}
```

The output looks like this:

Output

```
1) <nil>
panic: runtime error: invalid memory address or nil
pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8
pc=0x109ac86]

goroutine 1 [running]:
main.changeCreature(0x0)

/Users/corylanou/projects/learn/src/github.com/gopherguid
es/learn/_training/digital-ocean/pointers/src/nil.go:18
+0x26
    main.main()

/Users/corylanou/projects/learn/src/github.com/gopherguid
es/learn/_training/digital-ocean/pointers/src/nil.go:13
+0x98
        exit status 2
```

When we run the program, it printed out the value of the `creature` variable, and the value is `<nil>`. We then call the `changeCreature` function, and when that function tries to set the value of the `Species` field, it panics. This is because there is no instance of the variable actually created. Because of this, the program has no where to actually store the value, so the program panics.

It is common in Go that if you are receiving an argument as a pointer, you check to see if it was nil or not before performing any operations on it to prevent the program from panicking.

This is a common approach for checking for `nil`:

```
if someVariable == nil {
    // print an error or return from the method or fuction
}
```

Effectively you want to make sure you don't have a `nil` pointer that was passed into your function or method. If you do, you'll likely just want to return, or return an error to show that an invalid argument was passed to the function or method. The following code demonstrates checking for `nil`:

main.go

```go
package main

import "fmt"

type Creature struct {
	Species string
}

func main() {
	var creature *Creature

	fmt.Printf("1) %+v\n", creature)
	changeCreature(creature)
	fmt.Printf("3) %+v\n", creature)
}

func changeCreature(creature *Creature) {
	if creature == nil {
		fmt.Println("creature is nil")
		return
	}

	creature.Species = "jellyfish"
	fmt.Printf("2) %+v\n", creature)
}
```

We added a check in the `changeCreature` to see if the value of the `creature` argument was `nil`. If it was, we print out "creature is nil", and return out of the function. Otherwise, we continue and

change the value of the `Species` field. If we run the program, we will now get the following output:

Output

```
1) <nil>
creature is nil
3) <nil>
```

Notice that while we still had a `nil` value for the `creature` variable, we are no longer panicking because we are checking for that scenario.

Finally, if we create an instance of the `Creature` type and assign it to the `creature` variable, the program will now change the value as expected:

main.go

```
package main

import "fmt"

type Creature struct {
    Species string
}

func main() {
    var creature *Creature
    creature = &Creature{Species: "shark"}

    fmt.Printf("1) %+v\n", creature)
    changeCreature(creature)
    fmt.Printf("3) %+v\n", creature)
}

func changeCreature(creature *Creature) {
    if creature == nil {
        fmt.Println("creature is nil")
        return
    }

    creature.Species = "jellyfish"
    fmt.Printf("2) %+v\n", creature)
}
```

Now that we have an instance of the `Creature` type, the program will run and we will get the following expected output:

Output

```
1) &{Species:shark}
2) &{Species:jellyfish}
3) &{Species:jellyfish}
```

When you are working with pointers, there is a potential for the program to panic. To avoid panicking, you should check to see if a pointer value is `nil` prior to trying to access any of the fields or methods defined on it.

Next, let's look at how using pointers and values affects defining methods on a type.

Method Pointer Receivers

A receiver in go is the argument that is defined in a method declaration. Take a look at the following code:

```go
type Creature struct {
	Species string
}

func (c Creature) String() string {
	return c.Species
}
```

The receiver in this method is `c Creature`. It is stating that the instance of `c` is of type `Creature` and you will reference that type via that instance variable.

Just like the behavior of functions is different based on whether you send in an argument as a pointer or a value, methods also have different behavior. The big difference is that if you define a method with a value receiver, you are not able to make changes to the instance of that type that the method was defined on.

There will be times that you would like your method to be able to update the instance of the variable that you are using. To allow for this, you would want to make the receiver a pointer.

Let's add a `Reset` method to our `Creature` type that will set the `Species` field to an empty string:

main.go

```
package main

import "fmt"

type Creature struct {
    Species string
}

func (c Creature) Reset() {
    c.Species = ""
}

func main() {
    var creature Creature = Creature{Species: "shark"}

    fmt.Printf("1) %+v\n", creature)
    creature.Reset()
    fmt.Printf("2) %+v\n", creature)
}
```

If we run the program, we will get the following output:

Output

```
1) {Species:shark}
2) {Species:shark}
```

Notice that even though in the `Reset` method we set the value of `Species` to an empty string, that when we print out the value of our `creature` variable in the `main` function, the value is still set to `shark`. This is because we defined the `Reset` method has having a `value` receiver. This means that the method will only have access to a copy of the `creature` variable.

If we want to be able to modify the instance of the `creature` variable in the methods, we need to define them as having a `pointer` receiver:

main.go

```go
package main

import "fmt"

type Creature struct {
    Species string
}

func (c *Creature) Reset() {
    c.Species = ""
}

func main() {
    var creature Creature = Creature{Species: "shark"}

    fmt.Printf("1) %+v\n", creature)
    creature.Reset()
    fmt.Printf("2) %+v\n", creature)
}
```

Notice that we now added an asterisk (*) in front of the `Creature` type in when we defined the `Reset` method. This means that the instance of `Creature` that is passed to the `Reset` method is now a pointer, and as such when we make changes it will affect the original instance of that variables.

Output

```
1) {Species:shark}
2) {Species:}
```

The `Reset` method has now changed the value of the `Species` field.

Conclusion

Defining a function or method as a pass by value or pass by reference will affect what parts of your program are able to make changes to other parts. Controlling when that variable can be changed will allow you to write more robust and predictable software. Now that you have learned about pointers, you can see how they are used in interfaces as well.

Defining Structs in Go

Written by Gopher Guides

Building abstractions around concrete details is the greatest tool that a programming language can give to a developer. Structs allow Go developers to describe the world in which a Go program operates. Instead of reasoning about strings describing a `Street`, `City`, or a `PostalCode`, structs allow us to instead talk about an `Address`. They serve as a natural nexus for documentation in our efforts to tell future developers (ourselves included) what data is important to our Go programs and how future code should use that data appropriately. Structs can be defined and used in a few different ways. In this tutorial, we'll take a look at each of these techniques.

Defining Structs

Structs work like paper forms that you might use, for example, to file your taxes. Paper forms might have fields for textual pieces of information like your first and last names. Besides text fields, forms might have checkboxes to indicate Boolean values such as "married" or "single," or date fields for birth date. Similarly, structs collect different pieces of data together and organize them under different field names. When you initialize a variable with a new

struct, it’s as though you’ve photocopied a form and made it ready to fill out.

To create a new struct, you must first give Go a blueprint that describes the fields the struct contains. This struct definition usually begins with the keyword `type` followed by the name of the struct. After this, use the `struct` keyword followed by a pair of braces `{}` where you declare the fields the struct will contain. Once you have defined the struct, you are then able to declare variables that use this struct definition. This example defines a struct and uses it:

```
package main

import "fmt"

type Creature struct {
    Name string
}

func main() {
    c := Creature{
        Name: "Sammy the Shark",
    }
    fmt.Println(c.Name)
}
```

When you run this code, you will see this output:

output

```
Sammy the Shark
```

We first define a `Creature` struct in this example, containing a `Name` field of type `string`. Within the body of `main`, we create an instance of `Creature` by placing a pair of braces after the name of the type, `Creature`, and then specifying values for that instance's fields. The instance in `c` will have its `Name` field set to "Sammy the Shark". Within the `fmt.Println` function invocation, we retrieve the values of the instance's field by placing a period after the variable where the instance was created, followed by the name of the field we would like to access. For example, `c.Name` in this case returns the `Name` field.

When you declare a new instance of a struct, you generally enumerate the field names with their values, as in the last example. Alternatively, if every field value will be provided during the instantiation of a struct, you can omit the field names, like in this example:

```
package main

import "fmt"

type Creature struct {
    Name string
    Type string
}
```

```
func main() {
    c := Creature{"Sammy", "Shark"}
    fmt.Println(c.Name, "the", c.Type)
}
```

The output is the same as the last example:

output

```
Sammy the Shark
```

We’ve added an extra field to `Creature` to track the `Type` of creature as a `string`. When instantiating `Creature` within the body of `main`, we’ve opted to use the shorter instantiation form by providing values for each field in order and omitting their field names. In the declaration `Creature{"Sammy", "Shark"}`, the `Name` field takes the value `Sammy` and the `Type` field takes the value `Shark` because `Name` appears first in the type declaration, followed by `Type`.

This shorter declaration form has a few drawbacks that have led the Go community to prefer the longer form in most circumstances. You must provide values for each field in the struct when using the short declaration—you can’t skip fields you don’t care about. This quickly causes short declarations for structs with many fields to become confusing. For this reason, declaring structs using the short form is typically used with structs that have few fields.

The field names in the examples so far have all begun with capital letters. This is more significant than a stylistic preference. The use of capital or lowercase letters for field names affects whether your field names will be accessible to code running in other packages.

Struct Field Exporting

Fields of a struct follow the same exporting rules as other identifiers within the Go programming language. If a field name begins with a capital letter, it will be readable and writeable by code outside of the package where the struct was defined. If the field begins with a lowercase letter, only code within that struct’s package will be able to read and write that field. This example defines fields that are exported and those that are not:

```
package main

import "fmt"

type Creature struct {
    Name string
    Type string

    password string
}
```

```
func main() {
    c := Creature{
        Name: "Sammy",
        Type: "Shark",

        password: "secret",
    }
    fmt.Println(c.Name, "the", c.Type)
    fmt.Println("Password is", c.password)
}
```

This will output:

output

```
Sammy the Shark
Password is secret
```

We added an additional field to our previous examples, `secret`. `secret` is an unexported `string` field, which means that any other package that attempts to instantiate a `Creature` will not be able to access or set its `secret` field. Within the same package, we are able to access these fields, as this example has done. Since `main` is also in the `main` package, it's able to reference `c.password` and retrieve the value stored there. It's common to have unexported fields in structs with access to them mediated by exported methods.

Inline Structs

In addition to defining a new type to represent a struct, you can also define an inline struct. These on-the-fly struct definitions are useful in situations where inventing new names for struct types would be wasted effort. For example, tests often use a struct to define all the parameters that make up a particular test case. It would be cumbersome to come up with new names like `CreatureNamePrintingTestCase` when that struct is used in only one place.

Inline struct definitions appear on the right-hand side of a variable assignment. You must provide an instantiation of them immediately after by providing an additional pair of braces with values for each of the fields you define. The example that follows shows an inline struct definition:

```
package main

import "fmt"

func main() {
    c := struct {
        Name string
        Type string
    }{
        Name: "Sammy",
        Type: "Shark",
```

```
    }
    fmt.Println(c.Name, "the", c.Type)
}
```

The output from this example will be:

output

```
Sammy the Shark
```

Rather than defining a new type describing our struct with the `type` keyword, this example defines an inline struct by placing the `struct` definition immediately following the short-assignment operator, `:=`. We define the fields of the struct as in previous examples, but then we must immediately supply another pair of braces and the values that each field will assume. Using this struct is now exactly the same as before—we can refer to field names using dot notation. The most common place you will see inline structs declared is during tests, as frequently one-off structs are defined to contain data and expectations for a particular test case.

Conclusion

Structs are collections of heterogenous data defined by programmers to organize information. Most programs deal with enormous volumes of data, and without structs, it would become difficult to remember which `string` or `int`

variables belonged together or which were different. The next time that you find yourself juggling groups of variables, ask yourself if perhaps those variables would be better grouped together using a `struct`. Those variables may have been describing some higher-level concept all along.

}

Defining Methods in Go

Written by Gopher Guides

Functions allow you to organize logic into repeatable procedures that can use different arguments each time they run. In the course of defining functions, you'll often find that multiple functions might operate on the same piece of data each time. Go recognizes this pattern and allows you to define special functions, called methods, whose purpose is to operate on instances of some specific type, called a receiver. Adding methods to types allows you to communicate not only what the data is, but also how that data should be used.

Defining a Method

The syntax for defining a method is similar to the syntax for defining a function. The only difference is the addition of an extra parameter after the `func` keyword for specifying the receiver of the method. The receiver is a declaration of the type that you wish to define the method on. The following example defines a method on a struct type:

```go
package main

import "fmt"

type Creature struct {
    Name     string
    Greeting string
}

func (c Creature) Greet() {
    fmt.Printf("%s says %s", c.Name, c.Greeting)
}
```

```
func main() {
    sammy := Creature{
        Name:     "Sammy",
        Greeting: "Hello!",
    }
    Creature.Greet(sammy)
}
```

If you run this code, the output will be:

Output

```
Sammy says Hello!
```

We created a struct called `Creature` with `string` fields for a `Name` and a `Greeting`. This `Creature` has a single method defined, `Greet`. Within the receiver declaration, we assigned the instance of `Creature` to the variable `c` so that we could refer to the fields of the `Creature` as we assemble the greeting message in `fmt.Printf`.

In other languages, the receiver of method invocations is typically referred to by a keyword (e.g. `this` or `self`). Go considers the receiver to be a variable like any other, so you're free to name it whatever you like. The style preferred by the community for this parameter is a lower-case version of the first character of the receiver type. In this example, we used `c` because the receiver type was `Creature`.

Within the body of `main`, we created an instance of `Creature` and specified values for its `Name` and `Greeting` fields. We invoked the `Greet` method here by joining the name of the type and the name of the method with a `.` and supplying the instance of `Creature` as the first argument.

Go provides another, more convenient, way of calling methods on instances of a struct as shown in this example:

```
package main
```

```
import "fmt"

type Creature struct {
    Name     string
    Greeting string
}

func (c Creature) Greet() {
    fmt.Printf("%s says %s", c.Name, c.Greeting)
}

func main() {
    sammy := Creature{
        Name:     "Sammy",
        Greeting: "Hello!",
    }
    sammy.Greet()
}
```

If you run this, the output will be the same as the previous example:

Output

```
Sammy says Hello!
```

This example is identical to the previous one, but this time we have used dot notation to invoke the `Greet` method using the `Creature` stored in the `sammy` variable as the receiver. This is a shorthand notation for the function invocation in the first example. The standard library and the Go community prefers this style so much that you will rarely see the function invocation style shown earlier.

The next example shows one reason why dot notation is more prevalent:

```
package main

import "fmt"

type Creature struct {
    Name     string
    Greeting string
}

func (c Creature) Greet() Creature {
    fmt.Printf("%s says %s!\n", c.Name, c.Greeting)
    return c
}

func (c Creature) SayGoodbye(name string) {
    fmt.Println("Farewell", name, "!")
}

func main() {
    sammy := Creature{
        Name:     "Sammy",
        Greeting: "Hello!",
    }
    sammy.Greet().SayGoodbye("gophers")

    Creature.SayGoodbye(Creature.Greet(sammy), "gophers")
}
```

If you run this code, the output looks like this:

Output

```
Sammy says Hello!!
Farewell gophers !
Sammy says Hello!!
Farewell gophers !
```

We've modified the earlier examples to introduce another method called `SayGoodbye` and also changed `Greet` to return a `Creature` so that we can invoke further methods on that instance. In the body of `main` we call the methods `Greet` and `SayGoodbye` on the `sammy` variable first using dot notation and then using the functional invocation style.

Both styles output the same results, but the example using dot notation is far more readable. The chain of dots also tells us the sequence in which methods will be invoked, where the functional style inverts this sequence. The addition of a parameter to the `SayGoodbye` call further obscures the order of method calls. The clarity of dot notation is the reason that it is the preferred style for invoking methods in Go, both in the standard library and among the third-party packages you will find throughout the Go ecosystem.

Defining methods on types, as opposed to defining functions that operate on some value, have other special significance to the Go programming language. Methods are the core concept behind interfaces.

Interfaces

When you define a method on any type in Go, that method is added to the type's method set. The method set is the collection of functions associated with that type as methods and used by the Go compiler to determine whether some type can be assigned to a variable with an interface type. An interface type is a specification of methods used by the compiler to guarantee that a type provides implementations for those methods. Any type that has methods with the same name, same

parameters, and same return values as those found in an interface's definition are said to implement that interface and are allowed to be assigned to variables with that interface's type. The following is the definition of the `fmt.Stringer` interface from the standard library:

```
type Stringer interface {
  String() string
}
```

For a type to implement the `fmt.Stringer` interface, it needs to provide a `String()` method that returns a `string`. Implementing this interface will allow your type to be printed exactly as you wish (sometimes called "pretty-printed") when you pass instances of your type to functions defined in the `fmt` package. The following example defines a type that implements this interface:

```
package main

import (
    "fmt"
    "strings"
)

type Ocean struct {
    Creatures []string
}

func (o Ocean) String() string {
    return strings.Join(o.Creatures, ", ")
}

func log(header string, s fmt.Stringer) {
    fmt.Println(header, ":", s)
}
```

```go
func main() {
	o := Ocean{
		Creatures: []string{
			"sea urchin",
			"lobster",
			"shark",
		},
	}
	log("ocean contains", o)
}
```

When you run the code, you'll see this output:

Output

```
ocean contains : sea urchin, lobster, shark
```

This example defines a new struct type called `Ocean`. `Ocean` is said to implement the `fmt.Stringer` interface because `Ocean` defines a method called `String`, which takes no parameters and returns a `string`. In `main`, we defined a new `Ocean` and passed it to a `log` function, which takes a `string` to print out first, followed by anything that implements `fmt.Stringer`. The Go compiler allows us to pass `o` here because `Ocean` implements all of the methods requested by `fmt.Stringer`. Within `log`, we use `fmt.Println`, which calls the `String` method of `Ocean` when it encounters a `fmt.Stringer` as one of its parameters.

If `Ocean` did not provide a `String()` method, Go would produce a compilation error, because the `log` method requests a `fmt.Stringer` as its argument. The error looks like this:

Output

```
src/e4/main.go:24:6: cannot use o (type Ocean) as type
fmt.Stringer in argument to log:
        Ocean does not implement fmt.Stringer (missing String
method)
```

Go will also make sure that the `String()` method provided exactly matches the one requested by the `fmt.Stringer` interface. If it does not, it will produce an error that looks like this:

Output

```
src/e4/main.go:26:6: cannot use o (type Ocean) as type
fmt.Stringer in argument to log:
        Ocean does not implement fmt.Stringer (wrong type for
String method)
                have String()
                want String() string
```

In the examples so far, we have defined methods on the value receiver. That is, if we use the functional invocation of methods, the first parameter, referring to the type the method was defined on, will be a value of that type, rather than a pointer. Consequently, any modifications we make to the instance provided to the method will be discarded when the method completes execution, because the value received is a copy of the data. It's also possible to define methods on the pointer receiver to a type.

Pointer Receivers

The syntax for defining methods on the pointer receiver is nearly identical to defining methods on the value receiver. The difference is prefixing the name of the type in the receiver declaration with an

asterisk (*). The following example defines a method on the pointer receiver to a type:

```
package main

import "fmt"

type Boat struct {
    Name string

    occupants []string
}

func (b *Boat) AddOccupant(name string) *Boat {
    b.occupants = append(b.occupants, name)
    return b
}

func (b Boat) Manifest() {
    fmt.Println("The", b.Name, "has the following occupants:")
    for _, n := range b.occupants {
        fmt.Println("\t", n)
    }
}

func main() {
    b := &Boat{
        Name: "S.S. DigitalOcean",
    }

    b.AddOccupant("Sammy the Shark")
    b.AddOccupant("Larry the Lobster")
```

```
    b.Manifest()
}
```

You'll see the following output when you run this example:

Output

```
The S.S. DigitalOcean has the following occupants:
     Sammy the Shark
     Larry the Lobster
```

This example defined a `Boat` type with a `Name` and `occupants`. We want to force code in other packages to only add occupants with the `AddOccupant` method, so we've made the `occupants` field unexported by lowercasing the first letter of the field name. We also want to make sure that calling `AddOccupant` will cause the instance of `Boat` to be modified, which is why we defined `AddOccupant` on the pointer receiver. Pointers act as a reference to a specific instance of a type rather than a copy of that type. Knowing that `AddOccupant` will be invoked using a pointer to `Boat` guarantees that any modifications will persist.

Within `main`, we define a new variable, `b`, which will hold a pointer to a `Boat` (`*Boat`). We invoke the `AddOccupant` method twice on this instance to add two passengers. The `Manifest` method is defined on the `Boat` value, because in its definition, the receiver is specified as `(b Boat)`. In `main`, we are still able to call `Manifest` because Go is able to automatically dereference the pointer to obtain the `Boat` value. `b.Manifest()` here is equivalent to `(*b).Manifest()`.

Whether a method is defined on a pointer receiver or on a value receiver has important implications when trying to assign values to variables that are interface types.

Pointer Receivers and Interfaces

When you assign a value to a variable with an interface type, the Go compiler will examine the method set of the type being assigned to

ensure that it has the methods the interface expects. The method sets for the pointer receiver and the value receiver are different because methods that receive a pointer can modify their receiver where those that receive a value cannot.

The following example demonstrates defining two methods: one on a type's pointer receiver and on its value receiver. However, only the pointer receiver will be able to satisfy the interface also defined in this example:

```
package main

import "fmt"

type Submersible interface {
    Dive()
}

type Shark struct {
    Name string

    isUnderwater bool
}

func (s Shark) String() string {
    if s.isUnderwater {
        return fmt.Sprintf("%s is underwater", s.Name)
    }
    return fmt.Sprintf("%s is on the surface", s.Name)
}

func (s *Shark) Dive() {
    s.isUnderwater = true
}
```

```go
func submerge(s Submersible) {
    s.Dive()
}

func main() {
    s := &Shark{
        Name: "Sammy",
    }

    fmt.Println(s)

    submerge(s)

    fmt.Println(s)
}
```

When you run the code, you'll see this output:

Output

```
Sammy is on the surface
Sammy is underwater
```

This example defined an interface called `Submersible` that expects types having a `Dive()` method. We then defined a `Shark` type with a `Name` field and an `isUnderwater` method to keep track of the state of the `Shark`. We defined a `Dive()` method on the pointer receiver to `Shark` which modified `isUnderwater` to `true`. We also defined the `String()` method of the value receiver so that it could cleanly print the state of the `Shark` using `fmt.Println` by using the `fmt.Stringer` interface accepted by `fmt.Println` that we looked at earlier. We also used a function `submerge` that takes a `Submersible` parameter.

Using the `Submersible` interface rather than a `*Shark` allows the `submerge` function to depend only on the behavior provided by a type. This makes the `submerge` function more reusable because you wouldn't have to write new `submerge` functions for a `Submarine`, a `Whale`, or any other future aquatic inhabitants we haven't thought of yet. As long as they define a `Dive()` method, they can be used with the `submerge` function.

Within `main` we defined a variable `s` that is a pointer to a `Shark` and immediately printed `s` with `fmt.Println`. This shows the first part of the output, `Sammy is on the surface`. We passed `s` to `submerge` and then called `fmt.Println` again with `s` as its argument to see the second part of the output printed, `Sammy is underwater`.

If we changed `s` to be a `Shark` rather than a `*Shark`, the Go compiler would produce the error:

Output

```
cannot use s (type Shark) as type Submersible in argument to
submerge:
    Shark does not implement Submersible (Dive method has
pointer receiver)
```

The Go compiler helpfully tells us that `Shark` does have a `Dive` method, it's just defined on the pointer receiver. When you see this message in your own code, the fix is to pass a pointer to the interface type by using the & operator before the variable where the value type is assigned.

Conclusion

Declaring methods in Go is ultimately no different than defining functions that receive different types of variables. The same rules of working with pointers apply. Go provides some conveniences for this extremely common function definition and collects these into sets of

methods that can be reasoned about by interface types. Using methods effectively will allow you to work with interfaces in your code to improve testability and leaves better organization behind for future readers of your code.

If you'd like to learn more about the Go programming language in general, check out our How To Code in Go series.

How To Build and Install Go Programs

Written by Gopher Guides

So far in our How To Code in Go series, you have used the command `go run` to automatically compile your source code and run the resulting executable. Although this command is useful for testing your code on the command line, distributing or deploying your application requires you to build your code into a shareable binary executable, or a single file containing machine byte code that can run your application. To do this, you can use the Go toolchain to build and install your program.

In Go, the process of translating source code into a binary executable is called building. Once this executable is built, it will contain not only your application, but also all the support code needed to execute the binary on the target platform. This means that a Go binary does not need system dependencies such as Go tooling to run on a new system, unlike other languages like Ruby, Python, or Node.js. Putting these executables in an executable filepath on your own system will allow you to run the program from anywhere on your system. This is called installing the program onto your system.

In this tutorial, you will use the Go toolchain to run, build, and install a sample `Hello, World!` program, allowing you to use, distribute, and deploy future applications effectively.

Prerequisites

To follow the example in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment.

Setting Up and Running the Go Binary

First, create an application to use as an example for demonstrating the Go toolchain. To do this, you will use the classic “Hello, World!” program from the How To Write Your First Program in Go tutorial.

Create a directory called `greeter` in your `src` directory:

```
mkdir greeter
```

Next, move into the newly created directory and create the `main.go` file in the text editor of your choice:

```
cd greeter
nano main.go
```

Once the file is open, add the following contents:

src/greeter/main.go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

When run, this program will print the phrase `Hello, World!` to the console, and then the program will exit successfully.

Save and exit the file.

To test the program, use the `go run` command, as you've done in previous tutorials:

```
go run main.go
```

You'll receive the following output:

Output

```
Hello, World!
```

As mentioned before, the `go run` command built your source file into an executable binary, and then ran the compiled program. However, this tutorial aims to build the binary in such a way that you can share and distribute it at

will. To do this, you will use the `go build` command in the next step.

Building Go Binaries With `go build`

Using `go build`, you can generate an executable binary for our sample Go application, allowing you to distribute and deploy the program where you want.

Try this with `main.go`. In your `greeter` directory, run the following command:

```
go build
```

If you do not provide an argument to this command, `go build` will automatically compile the `main.go` program in your current directory. The command will include all your `*.go` files in the directory. It will also build all of the supporting code needed to be able to execute the binary on any computer with the same system architecture, regardless of whether that system has the `.go` source files, or even a Go installation.

In this case, you built your `greeter` application into an executable file that was added to your current directory. Check this by running the `ls` command:

```
ls
```

If you are running macOS or Linux, you will find a new executable file that has been named after the directory in which you built your program:

Output

```
greeter  main.go
```

Note: On Windows, your executable will be `greeter.exe`.

By default `go build` will generate an executable for the current platform and architecture. For example, if built on a `linux/386` system, the executable will be compatible with any other `linux/386` system, even if Go is not installed. Go supports building for other platforms and architectures, which you can read more about in our Building Go Applications for Different Operating Systems and Architectures article.

Now, that you've created your executable, run it to make sure the binary has been built correctly. On macOS or Linux, run the following command:

```
./greeter
```

On Windows, run:

```
greeter.exe
```

The output of the binary will match the output from when you ran the program with `go run`:

Output

```
Hello, World!
```

Now you have created a single executable binary that contains, not only your program, but also all of the system code needed to run that binary. You can now distribute this

program to new systems or deploy it to a server, knowing that the file will always run the same program.

In the next section, this tutorial will explain how a binary is named and how you can change it, so that you can have better control over the build process of your program.

Changing the Binary Name

Now that you know how to generate an executable, the next step is to identify how Go chooses a name for the binary and to customize this name for your project.

When you run `go build`, the default is for Go to automatically decide on the name of the generated executable. It does this in one of two ways: If you are using Go Modules, then Go will use the last part of your module's name; otherwise, Go will use the current directory's name. This is the method used in the last section, when you created the `greeter` directory, changed into it, and then ran `go build`.

Let's take a closer look at the module method. If you had a `go.mod` file in your project with a `module` declaration such as the following:

go.mod

```
module github.com/sammy/shark
```

Then the default name for the generated executable would be `shark`.

In more complex programs that require specific naming conventions, these default values will not always be the best choice for naming your binary. In these cases, it would be best to customize your output with the `-o` flag.

To test this out, change the name of the executable you made in the last section to `hello` and have it placed in a sub-folder called `bin`. You don't have to create this folder; Go will do that on its own during the build process.

Run the following `go build` command with the `-o` flag:

```
go build -o bin/hello
```

The `-o` flag makes Go match the output of the command to whatever argument you chose. In this case, the result is a new executable named `hello` in a sub-folder named `bin`.

To test the new executable, change into the new directory and run the binary:

```
cd bin
./hello
```

You will receive the following output:

Output

```
Hello, World!
```

You can now customize the name of your executable to fit the needs of your project, completing our survey of how to build binaries in Go. But with `go build`, you are still

limited to running your binary from the current directory. In order to use newly built executables from anywhere on your system, you can install it using `go install`.

Installing Go Programs with `go install`

So far in this article, we have discussed how to generate executable binaries from our `.go` source files. These executables are helpful to distribute, deploy, and test, but they cannot yet be executed from outside of their source directories. This would be a problem if you wanted to actively use your program, such as if you developed a command line tool to help your workflow on your own system. To make the programs easier to use, you can install them into your system and access them from anywhere.

To understand what is meant by this, you will use the `go install` command to install your sample application.

The `go install` command behaves almost identically to `go build`, but instead of leaving the executable in the current directory, or a directory specified by the `-o` flag, it places the executable into the `$GOPATH/bin` directory.

To find where your `$GOPATH` directory is located, run the following command:

```
go env GOPATH
```

The output you receive will vary, but the default is the `go` directory inside of your `$HOME` directory:

Output

```
$HOME/go
```

Since `go install` will place generated executables into a sub-directory of `$GOPATH` named `bin`, this directory must be added to the `$PATH` environment variable. This is covered in the Creating Your Go Workspace step of the prerequisite article [How To Install Go and Set Up a Local Programming Environment]().

With the `$GOPATH/bin` directory set up, move back to your `greeter` directory:

```
cd ..
```

Now run the install command:

```
go install
```

This will build your binary and place it in `$GOPATH/bin`. To test this, run the following:

```
ls $GOPATH/bin
```

This will list the contents of `$GOPATH/bin`:

Output

```
greeter
```

Note: The `go install` command does not support the `-o` flag, so it will use one of the default names described earlier to name the executable.

With the binary installed, test to see if the program will run from outside its source directory. Move back to your

home directory:

```
cd $HOME
```

Use the following to run the program:

```
greeter
```

This will yield the following:

Output

```
Hello, World!
```

Now you can take the programs you write and install them into your system, allowing you to use them from wherever, whenever you need them.

Conclusion

In this tutorial, you demonstrated how the Go toolchain makes it easy to build executable binaries from source code. These binaries can be distributed to run on other systems, even ones that do not have Go tooling and environments. You also used `go install` to automatically build and install our programs as executables in the system's `$PATH`. With `go build` and `go install`, you now have the ability to share and use your application at will.

Now that you know the basics of `go build`, you can explore how to make modular source code with the Customizing Go Binaries with Build Tags tutorial, or how to build for different platforms with Building Go Applications

for Different Operating Systems and Architectures. If you’d like to learn more about the Go programming language in general, check out the entire How To Code in Go series.

How To Use Struct Tags in Go

Written by Gopher Guides

Structures, or structs, are used to collect multiple pieces of information together in one unit. These collections of information are used to describe higher-level concepts, such as an `Address` composed of a `Street`, `City`, `State`, and `PostalCode`. When you read this information from systems such as databases, or APIs, you can use struct tags to control how this information is assigned to the fields of a struct. Struct tags are small pieces of metadata attached to fields of a struct that provide instructions to other Go code that works with the struct.

What Does a Struct Tag Look Like?

Go struct tags are annotations that appear after the type in a Go struct declaration. Each tag is composed of short strings associated with some corresponding value.

A struct tag looks like this, with the tag offset with backtick ` characters:

```
type User struct {
    Name string `example:"name"`
}
```

Other Go code is then capable of examining these structs and extracting the values assigned to specific keys it requests. Struct tags have no effect on the operation of your code without some other code that examines them.

Try this example to see what struct tags look like, and that without code from another package, they will have no effect.

```
package main
```

```go
import "fmt"

type User struct {
    Name string `example:"name"`
}

func (u *User) String() string {
    return fmt.Sprintf("Hi! My name is %s", u.Name)
}

func main() {
    u := &User{
        Name: "Sammy",
    }

    fmt.Println(u)
}
```

This will output:

Output

```
Hi! My name is Sammy
```

This example defines a `User` type with a `Name` field. The `Name` field has been given a struct tag of `example:"name"`. We would refer to this specific tag in conversation as the "example struct tag" because it uses the word "example" as its key. The `example` struct tag has the value `"name"` for the `Name` field. On the `User` type, we also define the `String()` method required by the `fmt.Stringer` interface. This will be called automatically when we pass the type to `fmt.Println` and gives us a chance to produce a nicely formatted version of our struct.

Within the body of `main`, we create a new instance of our `User` type and pass it to `fmt.Println`. Even though the struct had a struct tag present, we see that it has no effect on the operation of this Go code. It will behave exactly the same if the struct tag were not present.

To use struct tags to accomplish something, other Go code must be written to examine structs at runtime. The standard library has packages that use struct tags as part of their operation. The most popular of these is the `encoding/json` package.

Encoding JSON

JavaScript Object Notation (JSON) is a textual format for encoding collections of data organized under different string keys. It's commonly used to communicate data between different programs as the format is simple enough that libraries exist to decode it in many different languages. The following is an example of JSON:

```
{
  "language": "Go",
  "mascot": "Gopher"
}
```

This JSON object contains two keys, `language` and `mascot`. Following these keys are the associated values. Here the `language` key has a value of `Go` and `mascot` is assigned the value `Gopher`.

The JSON encoder in the standard library makes use of struct tags as annotations indicating to the encoder how you would like to name your fields in the JSON output. These JSON encoding and decoding mechanisms can be found in the `encoding/json` package.

Try this example to see how JSON is encoded without struct tags:

```
package main

import (
    "encoding/json"
```

```go
    "fmt"
    "log"
    "os"
    "time"
)

type User struct {
    Name          string
    Password      string
    PreferredFish []string
    CreatedAt     time.Time
}

func main() {
    u := &User{
        Name:      "Sammy the Shark",
        Password:  "fisharegreat",
        CreatedAt: time.Now(),
    }

    out, err := json.MarshalIndent(u, "", "   ")
    if err != nil {
        log.Println(err)
        os.Exit(1)
    }

    fmt.Println(string(out))
}
```

This will print the following output:

Output

```
{
  "Name": "Sammy the Shark",
  "Password": "fisharegreat",
  "CreatedAt": "2019-09-23T15:50:01.203059-04:00"
}
```

We defined a struct describing a user with fields including their name, password, and the time the user was created. Within the `main` function, we create an instance of this user by supplying values for all fields except `PreferredFish` (Sammy likes all fish). We then passed the instance of `User` to the `json.MarshalIndent` function. This is used so we can more easily see the JSON output without using an external formatting tool. This call could be replaced with `json.Marshal(u)` to receive JSON without any additional whitespace. The two additional arguments to `json.MarshalIndent` control the prefix to the output (which we have omitted with the empty string), and the characters to use for indenting, which here are two space characters. Any errors produced from `json.MarshalIndent` are logged and the program terminates using `os.Exit(1)`. Finally, we cast the `[]byte` returned from `json.MarshalIndent` to a `string` and hand the resulting string to `fmt.Println` for printing on the terminal.

The fields of the struct appear exactly as we named them. This is not the typical JSON style that you may expect, which uses camel casing for names of fields. You'll change the names of the field to follow camel case style in this next example. As you'll see when you run this example, this won't work because the desired field names conflict with Go's rules about exported field names.

```
package main

import (
```

```
    "encoding/json"
    "fmt"
    "log"
    "os"
    "time"
)

type User struct {
    name          string
    password      string
    preferredFish []string
    createdAt     time.Time
}

func main() {
    u := &User{
        name:      "Sammy the Shark",
        password:  "fisharegreat",
        createdAt: time.Now(),
    }

    out, err := json.MarshalIndent(u, "", "  ")
    if err != nil {
        log.Println(err)
        os.Exit(1)
    }

    fmt.Println(string(out))
}
```

This will present the following output:

Output

```
{}
```

In this version, we've altered the names of the fields to be camel cased. Now `Name` is `name`, `Password` is `password`, and finally `CreatedAt` is `createdAt`. Within the body of `main` we've changed the instantiation of our struct to use these new names. We then pass the struct to the `json.MarshalIndent` function as before. The output, this time is an empty JSON object, `{}`.

Camel casing fields properly requires that the first character be lower-cased. While JSON doesn't care how you name your fields, Go does, as it indicates the visibility of the field outside of the package. Since the `encoding/json` package is a separate package from the `main` package we're using, we must uppercase the first character in order to make it visible to `encoding/json`. It would seem that we're at an impasse, and we need some way to convey to the JSON encoder what we would like this field to be named.

Using Struct Tags to Control Encoding

You can modify the previous example to have exported fields that are properly encoded with camel-cased field names by annotating each field with a struct tag. The struct tag that `encoding/json` recognizes has a key of `json` and a value that controls the output. By placing the camel-cased version of the field names as the value to the `json` key, the encoder will use that name instead. This example fixes the previous two attempts:

```go
package main

import (
    "encoding/json"
    "fmt"
```

```
    "log"
    "os"
    "time"
)

type User struct {
    Name          string    `json:"name"`
    Password      string    `json:"password"`
    PreferredFish []string  `json:"preferredFish"`
    CreatedAt     time.Time `json:"createdAt"`
}

func main() {
    u := &User{
        Name:      "Sammy the Shark",
        Password:  "fisharegreat",
        CreatedAt: time.Now(),
    }

    out, err := json.MarshalIndent(u, "", "   ")
    if err != nil {
        log.Println(err)
        os.Exit(1)
    }

    fmt.Println(string(out))
}
```

This will output:

Output

```
{
  "name": "Sammy the Shark",
  "password": "fisharegreat",
  "preferredFish": null,
  "createdAt": "2019-09-23T18:16:17.57739-04:00"
}
```

We've changed the field names back to be visible to other packages by capitalizing the first letters of their names. However, this time we've added struct tags in the form of `json:"name"`, where `"name"` was the name we wanted `json.MarshalIndent` to use when printing our struct as JSON.

We've now successfully formatted our JSON correctly. Notice, however, that the fields for some values were printed even though we did not set those values. The JSON encoder can eliminate these fields as well, if you like.

Removing Empty JSON Fields

Most commonly, we want to suppress outputting fields that are unset in JSON. Since all types in Go have a "zero value," some default value that they are set to, the `encoding/json` package needs additional information to be able to tell that some field should be considered unset when it assumes this zero value. Within the value part of any `json` struct tag, you can suffix the desired name of your field with `,omitempty` to tell the JSON encoder to suppress the output of this field when the field is set to the zero value. The following example fixes the previous examples to no longer output empty fields:

```go
package main

import (
```

```go
    "encoding/json"
    "fmt"
    "log"
    "os"
    "time"
)

type User struct {
    Name          string    `json:"name"`
    Password      string    `json:"password"`
    PreferredFish []string  `json:"preferredFish,omitempty"`
    CreatedAt     time.Time `json:"createdAt"`
}

func main() {
    u := &User{
        Name:      "Sammy the Shark",
        Password:  "fisharegreat",
        CreatedAt: time.Now(),
    }

    out, err := json.MarshalIndent(u, "", "  ")
    if err != nil {
        log.Println(err)
        os.Exit(1)
    }

    fmt.Println(string(out))
}
```

This example will output:

Output

```
{
  "name": "Sammy the Shark",
  "password": "fisharegreat",
  "createdAt": "2019-09-23T18:21:53.863846-04:00"
}
```

We’ve modified the previous examples so that the `PreferredFish` field now has the struct tag `json:"preferredFish,omitempty"`. The presence of the `,omitempty` augmentation causes the JSON encoder to skip that field, since we decided to leave it unset. This had the value `null` in our previous examples’ outputs.

This output is looking much better, but we’re still printing out the user’s password. The `encoding/json` package provides another way for us to ignore private fields entirely.

Ignoring Private Fields

Some fields must be exported from structs so that other packages can correctly interact with the type. However, the nature of these fields may be sensitive, so in these circumstances, we would like the JSON encoder to ignore the field entirely—even when it is set. This is done using the special value `-` as the value argument to a `json:` struct tag.

This example fixes the issue of exposing the user’s password.

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "os"
    "time"
```

```
)

type User struct {
    Name      string    `json:"name"`
    Password  string    `json:"-"`
    CreatedAt time.Time `json:"createdAt"`
}

func main() {
    u := &User{
        Name:      "Sammy the Shark",
        Password:  "fisharegreat",
        CreatedAt: time.Now(),
    }

    out, err := json.MarshalIndent(u, "", "  ")
    if err != nil {
        log.Println(err)
        os.Exit(1)
    }

    fmt.Println(string(out))
}
```

When you run this example, you'll see this output:

Output

```
{
  "name": "Sammy the Shark",
  "createdAt": "2019-09-23T16:08:21.124481-04:00"
}
```

The only thing we've changed in this example from previous ones is that the password field now uses the special `"-"` value for its `json:` struct tag. We see that in the output from this example that the `password` field is no longer present.

These features of the `encoding/json` package, `,omitempty` and `"-"`, are not standards. What a package decides to do with values of a struct tag depends on its implementation. Because the `encoding/json` package is part of the standard library, other packages have also implemented these features in the same way as a matter of convention. However, it's important to read the documentation for any third-party package that uses struct tags to learn what is supported and what is not.

Conclusion

Struct tags offer a powerful means to augment the functionality of code that works with your structs. Many standard library and third-party packages offer ways to customize their operation through the use of struct tags. Using them effectively in your code provides both this customization behavior and succinctly documents how these fields are used to future developers.

How To Use Interfaces in Go

Written by Gopher Guides

Writing flexible, reusable, and modular code is vital for developing versatile programs. Working in this way ensures code is easier to maintain by avoiding the need to make the same change in multiple places. How you accomplish this varies from language to language. For instance, inheritance is a common approach that is used in languages such as Java, C++, C#, and more.

Developers can also attain those same design goals through composition. Composition is a way to combine objects or data types into more complex ones. This is the approach that Go uses to promote code reuse, modularity, and flexibility. Interfaces in Go provide a method of organizing complex compositions, and learning how to use them will allow you to create common, reusable code.

In this article, we will learn how to compose custom types that have common behaviors, which will allow us to reuse our code. We'll also learn how to implement interfaces for our own custom types that will satisfy interfaces defined from another package.

Defining a Behavior

One of the core implementations of composition is the use of interfaces. An interface defines a behavior of a type. One of the most commonly used interfaces in the Go standard library is the `fmt.Stringer` interface:

```
type Stringer interface {
    String() string
}
```

The first line of code defines a `type` called `Stringer`. It then states that it is an `interface`. Just like defining a struct, Go uses curly braces (`{}`) to surround the definition of the interface. In comparison to defining structs, we only define the interface's behavior; that is, "what can this type do".

In the case of the `Stringer` interface, the only behavior is the `String()` method. The method takes no arguments and returns a string.

Next, let's look at some code that has the `fmt.Stringer` behavior:

main.go

```
package main

import "fmt"

type Article struct {
    Title string
    Author string
}

func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title, a.Author
}

func main() {
    a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
    }
    fmt.Println(a.String())
}
```

The first thing we do is create a new type called `Article`. This type has a `Title` and an `Author` field and both are of the string data type:

main.go

```
...
type Article struct {
    Title string
    Author string
}
...
```

Next, we define a method called `String` on the `Article` type. The `String` method will return a string that represents the `Article` type:

main.go

```
...
func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title, a.Author
}
...
```

Then, in our main function, we create an instance of the Article type and assign it to the variable called a. We provide the values of "Understanding Interfaces in Go" for the Title field, and "Sammy Shark" for the Author field:

main.go

```
...
a := Article{
    Title: "Understanding Interfaces in Go",
    Author: "Sammy Shark",
}
...
```

Then, we print out the result of the String method by calling fmt.Println and passing in the result of the a.String() method call:

main.go

```
...
fmt.Println(a.String())
```

After running the program you'll see the following output:

Output

```
The "Understanding Interfaces in Go" article was written by Sammy Shark.
```

So far, we haven't used an interface, but we did create a type that had a behavior. That behavior matched the fmt.Stringer interface. Next, let's see how we can use that behavior to make our code more reusable.

Defining an Interface

Now that we have our type defined with the desired behavior, we can look at how to use that behavior.

Before we do that, however, let's look at what we would need to do if we wanted to call the String method from the Article type in a function:

main.go

```
package main

import "fmt"

type Article struct {
    Title string
    Author string
}

func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}

func main() {
    a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
    }
    Print(a)
}

func Print(a Article) {
    fmt.Println(a.String())
}
```

In this code we add a new function called `Print` that takes an `Article` as an argument. Notice that the only thing the `Print` function does is call the `String` method. Because of this, we could instead define an interface to pass to the function:

main.go

```
package main

import "fmt"

type Article struct {
    Title string
    Author string
}

func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}

type Stringer interface {
    String() string
}

func main() {
    a := Article{
        Title: "Understanding Interfaces in Go",
        Author: "Sammy Shark",
    }
    Print(a)
}

func Print(s Stringer) {
    fmt.Println(s.String())
}
```

Here we created an interface called `Stringer`:

main.go

```
...
type Stringer interface {
    String() string
}
...
```

The `Stringer` interface has only one method, called `String()` that returns a `string`. A method is a special function that is scoped to a specific type in Go. Unlike a function, a method can only be called from the instance of the type it was defined on.

We then update the signature of the `Print` method to take a `Stringer`, and not a concrete type of `Article`. Because the compiler knows that a `Stringer` interface defines the `String` method, it will only accept types that also have the `String` method.

Now we can use the `Print` method with anything that satisfies the `Stringer` interface. Let's create another type to demonstrate this:

main.go

```
package main

import "fmt"

type Article struct {
    Title  string
    Author string
}

func (a Article) String() string {
    return fmt.Sprintf("The %q article was written by %s.", a.Title,
a.Author)
}

type Book struct {
    Title  string
    Author string
    Pages  int
}

func (b Book) String() string {
    return fmt.Sprintf("The %q book was written by %s.", b.Title, b.Author)
}

type Stringer interface {
    String() string
}

func main() {
    a := Article{
        Title:  "Understanding Interfaces in Go",
        Author: "Sammy Shark",
    }
    Print(a)

    b := Book{
        Title:  "All About Go",
        Author: "Jenny Dolphin",
        Pages:  25,
```

```
    }
    Print(b)
}

func Print(s Stringer) {
    fmt.Println(s.String())
}
```

We now add a second type called `Book`. It also has the `String` method defined. This means it also satisfies the `Stringer` interface. Because of this, we can also send it to our `Print` function:

Output

```
The "Understanding Interfaces in Go" article was written by Sammy Shark.
The "All About Go" book was written by Jenny Dolphin. It has 25 pages.
```

So far, we have demonstrated how to use just a single interface. However, an interface can have more than one behavior defined. Next, we'll see how we can make our interfaces more versatile by declaring more methods.

Multiple Behaviors in an Interface

One of the core tenants of writing Go code is to write small, concise types and compose them up to larger, more complex types. The same is true when composing interfaces. To see how we build up an interface, we'll first start by defining only one interface. We'll define two shapes, a `Circle` and `Square`, and they will both define a method called `Area`. This method will return the geometric area of their respective shapes:

main.go

```go
package main

import (
	"fmt"
	"math"
)

type Circle struct {
	Radius float64
}

func (c Circle) Area() float64 {
	return math.Pi * math.Pow(c.Radius, 2)
}

type Square struct {
	Width  float64
	Height float64
}

func (s Square) Area() float64 {
	return s.Width * s.Height
}

type Sizer interface {
	Area() float64
}

func main() {
	c := Circle{Radius: 10}
	s := Square{Height: 10, Width: 5}

	l := Less(c, s)
	fmt.Printf("%+v is the smallest\n", l)
}

func Less(s1, s2 Sizer) Sizer {
	if s1.Area() < s2.Area() {
		return s1
```

```
    }
    return s2
}
```

Because each type declares the Area method, we can create an interface that defines that behavior. We create the following Sizer interface:

main.go

```
...
type Sizer interface {
    Area() float64
}
...
```

We then define a function called Less that takes two Sizer and returns the smallest one:

main.go

```
...
func Less(s1, s2 Sizer) Sizer {
    if s1.Area() < s2.Area() {
        return s1
    }
    return s2
}
...
```

Notice that we not only accept both arguments as the type Sizer, but we also return the result as a Sizer as well. This means that we no longer return a Square or a Circle, but the interface of Sizer.

Finally, we print out what had the smallest area:

Output

```
{Width:5 Height:10} is the smallest
```

Next, let's add another behavior to each type. This time we'll add the String() method that returns a string. This will satisfy the fmt.Stringer interface:

main.go

```
package main

import (
	"fmt"
	"math"
)

type Circle struct {
	Radius float64
}

func (c Circle) Area() float64 {
	return math.Pi * math.Pow(c.Radius, 2)
}

func (c Circle) String() string {
	return fmt.Sprintf("Circle {Radius: %.2f}", c.Radius)
}

type Square struct {
	Width  float64
	Height float64
}

func (s Square) Area() float64 {
	return s.Width * s.Height
}

func (s Square) String() string {
	return fmt.Sprintf("Square {Width: %.2f, Height: %.2f}", s.Width,
s.Height)
}

type Sizer interface {
	Area() float64
}

type Shaper interface {
	Sizer
```

```
    fmt.Stringer
}

func main() {
    c := Circle{Radius: 10}
    PrintArea(c)

    s := Square{Height: 10, Width: 5}
    PrintArea(s)

    l := Less(c, s)
    fmt.Printf("%v is the smallest\n", l)

}

func Less(s1, s2 Sizer) Sizer {
    if s1.Area() < s2.Area() {
        return s1
    }
    return s2
}

func PrintArea(s Shaper) {
    fmt.Printf("area of %s is %.2f\n", s.String(), s.Area())
}
```

Because both the `Circle` and the `Square` type implement both the `Area` and `String` methods, we can now create another interface to describe that wider set of behavior. To do this, we'll create an interface called `Shaper`. We'll compose this of the `Sizer` interface and the `fmt.Stringer` interface:

main.go

```
...
type Shaper interface {
    Sizer
    fmt.Stringer
}
...
```

Note: It is considered idiomatic to try to name your interface by ending in `er`, such as `fmt.Stringer`, `io.Writer`, etc. This is why we named our interface `Shaper`, and not `Shape`.

Now we can create a function called `PrintArea` that takes a `Shaper` as an argument. This means that we can call both methods on the passed in value for both the `Area` and `String` method:

main.go

```
...
func PrintArea(s Shaper) {
    fmt.Printf("area of %s is %.2f\n", s.String(), s.Area())
}
```

If we run the program, we will receive the following output:

Output

```
area of Circle {Radius: 10.00} is 314.16
area of Square {Width: 5.00, Height: 10.00} is 50.00
Square {Width: 5.00, Height: 10.00} is the smallest
```

We have now seen how we can create smaller interfaces and build them up into larger ones as needed. While we could have started with the larger interface and passed it to all of our functions, it is considered best practice to send only the smallest interface to a function that is needed. This typically results in clearer code, as anything that accepts a specific smaller interface only intends to work with that defined behavior.

For example, if we passed `Shaper` to the `Less` function, we may assume that it is going to call both the `Area` and `String` methods. However, since we only intend to call the `Area` method, it makes the `Less` function clear as we know that we can only call the `Area` method of any argument passed to it.

Conclusion

We have seen how creating smaller interfaces and building them up to larger ones allows us to share only what we need to a function or method. We also learned that we can compose our interfaces from other interfaces, including those defined from other packages, not just our packages.

If you'd like to learn more about the Go programming language, check out the entire How To Code in Go series.

Building Go Applications for Different Operating Systems and Architectures

Written by Gopher Guides

In software development, it is important to consider the operating system and underlying processor architecture that you would like to compile your binary for. Since it is often slow or impossible to run a binary on a different OS/architecture platform, it is a common practice to build your final binary for many different platforms to maximize your program's audience. However, this can be difficult when the platform you are using for development is different from the platform you want to deploy your program to. In the past, for example, developing a program on Windows and deploying it to a Linux or a macOS machine would involve setting up build machines for each of the environments you wanted binaries for. You'd also need to keep your tooling in sync, in addition to other considerations that would add cost and make collaborative testing and distribution more difficult.

Go solves this problem by building support for multiple platforms directly into the `go build` tool, as well as the rest of the Go toolchain. By using environment variables and build tags, you can control which OS and architecture your final binary is built for, in addition to putting together a workflow that can quickly toggle the inclusion of platform-dependent code without changing your codebase.

In this tutorial, you will put together a sample application that joins strings together into a filepath, create and selectively include platform-dependent snippets, and build binaries for multiple operating systems and system architectures on your own system, showing you how to use this powerful capability of the Go programming language.

Prerequisites

To follow the example in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment.

Possible Platforms for `GOOS` and `GOARCH`

Before showing how to control the build process to build binaries for different platforms, let's first inspect what kinds of platforms Go is capable of building for, and how Go references these platforms using the environment variables `GOOS` and `GOARCH`.

The Go tooling has a command that can print a list of the possible platforms that Go can build on. This list can change with each new Go release, so the combinations discussed here might not be the same on another version of Go. At the time of writing this tutorial, the current Go release is `1.13`.

To find this list of possible platforms, run the following:

```
go tool dist list
```

You will receive an output similar to the following:

Output

```
aix/ppc64        freebsd/amd64   linux/mipsle   openbsd/386
android/386      freebsd/arm     linux/ppc64    openbsd/amd64
android/amd64    illumos/amd64   linux/ppc64le  openbsd/arm
android/arm      js/wasm         linux/s390x    openbsd/arm64
android/arm64    linux/386       nacl/386       plan9/386
darwin/386       linux/amd64     nacl/amd64p32  plan9/amd64
darwin/amd64     linux/arm       nacl/arm       plan9/arm
darwin/arm       linux/arm64     netbsd/386     solaris/amd64
darwin/arm64     linux/mips      netbsd/amd64   windows/386
dragonfly/amd64  linux/mips64    netbsd/arm     windows/amd64
freebsd/386      linux/mips64le  netbsd/arm64   windows/arm
```

This output is a set of key-value pairs separated by a `/`. The first part of the combination, before the `/`, is the operating system. In Go, these operating systems are possible values for the environment variable `GOOS`, pronounced "goose", which stands for Go Operating System. The second part, after the `/`, is the architecture. As before, these are all possible values for an environment variable: `GOARCH`. This is pronounced "gore-ch", and stands for Go Architecture.

Let's break down one of these combinations to understand what it means and how it works, using `linux/386` as an example. The key-value pair starts with the `GOOS`, which in this example would be `linux`, referring to the Linux OS. The `GOARCH` here would be `386`, which stands for the Intel 80386 microprocessor.

There are many platforms available with the `go build` command, but a majority of the time you'll end up using `linux` , `windows`, or `darwin` as a value for `GOOS`. These cover the big three OS platforms: Linux, Windows, and macOS, which is based on the Darwin operating system and is thus called `darwin`. However, Go can also cover less mainstream platforms like `nacl`, which represents Google's Native Client.

When you run a command like `go build`, Go uses the current platform's `GOOS` and `GOARCH` to determine how to build the binary. To find out what combination your platform is, you can use the `go env` command and pass `GOOS` and `GOARCH` as arguments:

```
go env GOOS GOARCH
```

In testing this example, we ran this command on macOS on a machine with an AMD64 architecture, so we will receive the following output:

Output

```
darwin
amd64
```

Here the output of the command tells us that our system has `GOOS=darwin` and `GOARCH=amd64`.

You now know what the `GOOS` and `GOARCH` are in Go, as well as their possible values. Next, you will put together a program to use as an example of how to use these environment variables and build tags to build binaries for other platforms.

Write a Platform-Dependent Program with `filepath.Join()`

Before you start building binaries for other platforms, let's build an example program. A good sample for this purpose is the `Join` function in the [`path/filepath`] package in the Go standard library. This function takes a number of strings and returns one string that is joined together with the correct filepath separator.

This is a good example program because the operation of the program depends on which OS it is running on. On Windows, the path separator is a backslash, `\`, while Unix-based systems use a forward slash, `/`.

Let's start with building an application that uses `filepath.Join()`, and later, you'll write your own implementation of the `Join()` function that customizes the code to the platform-specific binaries.

First, create a folder in your `src` directory with the name of your app:

```
mkdir app
```

Move into that directory:

```
cd app
```

Next, create a new file in your text editor of choice named `main.go`. For this tutorial, we will use Nano:

```
nano main.go
```

Once the file is open, add the following code:

src/app/main.go

```
package main

import (
  "fmt"
  "path/filepath"
)

func main() {
  s := filepath.Join("a", "b", "c")
  fmt.Println(s)
}
```

The `main()` function in this file uses `filepath.Join()` to concatenate three strings together with the correct, platform-dependent path separator.

Save and exit the file, then run the program:

```
go run main.go
```

When running this program, you will receive different output depending on which platform you are using. On Windows, you will see the strings separated by `\`:

Output

```
a\b\c
```

On Unix systems like macOS and Linux, you will receive the following:

Output

```
a/b/c
```

This shows that, because of the different filesystem protocols used on these operating systems, the program will have to build different code for the different platforms. But since it already uses a different file separator depending on the OS, we know that `filepath.Join()` already accounts for the difference in platform. This is because the Go tool chain automatically detects your machine's `GOOS` and `GOARCH` and uses this information to use the code snippet with the right build tags and file separator.

Let's consider where the `filepath.Join()` function gets its separator from. Run the following command to inspect the relevant snippet from Go's standard library:

```
less /usr/local/go/src/os/path_unix.go
```

This will display the contents of `path_unix.go`. Look for the following part of the file:

/usr/local/go/os/path_unix.go

```
. . .
// +build aix darwin dragonfly freebsd js,wasm linux nacl netbsd openbsd sola

package os

const (
  PathSeparator     = '/' // OS-specific path separator
  PathListSeparator = ':' // OS-specific path list separator
)
. . .
```

This section defines the `PathSeparator` for all of the varieties of Unix-like systems that Go supports. Notice all of the build tags at the top, which are each one of the possible Unix `GOOS` platforms associated with Unix. When the `GOOS` matches these terms, your program will yield the Unix-styled filepath separator.

Press `q` to return to the command line.

Next, open the file that defines the behavior of `filepath.Join()` when used on Windows:

```
less /usr/local/go/src/os/path_windows.go
```

You will see the following:

/usr/local/go/os/path_unix.go

```
. . .
package os

const (
        PathSeparator     = '\\' // OS-specific path separator
        PathListSeparator = ';'  // OS-specific path list separator
)
. . .
```

Although the value of `PathSeparator` is `\\` here, the code will render the single backslash (\) needed for Windows filepaths, since the first backslash is only needed as an escape character.

Notice that, unlike the Unix file, there are no build tags at the top. This is because `GOOS` and `GOARCH` can also be passed to `go build` by adding an underscore (_) and the environment variable value as a suffix to the filename, something we will go into more in the section Using GOOS and GOARCH File Name Suffixes. Here, the `_windows` part of `path_windows.go` makes the file act as if it had the build tag `// +build windows` at the

top of the file. Because of this, when your program is run on Windows, it will use the constants of `PathSeparator` and `PathListSeparator` from the `path_windows.go` code snippet.

To return to the command line, quit `less` by pressing `q`.

In this step, you built a program that showed how Go converts the `GOOS` and `GOARCH` automatically into build tags. With this in mind, you can now update your program and write your own implementation of `filepath.Join()`, using build tags to manually set the correct `PathSeparator` for Windows and Unix platforms.

Implementing a Platform-Specific Function

Now that you know how Go's standard library implements platform-specific code, you can use build tags to do this in your own **app** program. To do this, you will write your own implementation of `filepath.Join()`.

Open up your `main.go` file:

```
nano main.go
```

Replace the contents of `main.go` with the following, using your own function called `Join()`:

src/app/main.go

```go
package main

import (
  "fmt"
  "strings"
)

func Join(parts ...string) string {
  return strings.Join(parts, PathSeparator)
}

func main() {
  s := Join("a", "b", "c")
  fmt.Println(s)
}
```

The `Join` function takes a number of `parts` and joins them together using the `strings.Join()` method from the `strings package` to concatenate the `parts` together using the `PathSeparator`.

You haven't defined the `PathSeparator` yet, so do that now in another file. Save and quit `main.go`, open your favorite editor, and create a new file named `path.go`:

```
nano path.go
```

Define the `PathSeparator` and set it equal to the Unix filepath separator, `/`:

src/app/path.go

```
package main

const PathSeparator = "/"
```

Compile and run the application:

```
go build
./app
```

You'll receive the following output:

Output

```
a/b/c
```

This runs successfully to get a Unix-style filepath. But this isn't yet what we want: the output is always `a/b/c`, regardless of what platform it runs on. To add in the functionality to create Windows-style filepaths, you will need to add a Windows version of the `PathSeparator` and tell the `go build` command which version to use. In the next section, you will use build tags to accomplish this.

Using GOOS or GOARCH Build Tags

To account for Windows platforms, you will now create an alternate file to `path.go` and use build tags to make sure the code snippets only run when `GOOS` and `GOARCH` are the appropriate platform.

But first, add a build tag to `path.go` to tell it to build for everything except for Windows. Open up the file:

```
nano path.go
```

Add the following highlighted build tag to the file:

src/app/path.go

```
// +build !windows

package main

const PathSeparator = "/"
```

Go build tags allow for inverting, meaning that you can instruct Go to build this file for any platform except for Windows. To invert a build tag, place a `!` before the tag.

Save and exit the file.

Now, if you were to run this program on Windows, you would get the following error:

Output

```
./main.go:9:29: undefined: PathSeparator
```

In this case, Go would not be able to include `path.go` to define the variable `PathSeparator`.

Now that you have ensured that `path.go` will not run when `GOOS` is Windows, add a new file, `windows.go`:

```
nano windows.go
```

In `windows.go`, define the Windows `PathSeparator`, as well as a build tag to let the `go build` command know it is the Windows implementation:

src/app/windows.go

```go
// +build windows

package main

const PathSeparator = "\\"
```

Save the file and exit from the text editor. The application can now compile one way for Windows and another for all other platforms.

While the binaries will now build correctly for their platforms, there are further changes you must make in order to compile for a platform that you do not have access to. To do this, you will alter your local `GOOS` and `GOARCH` environment variables in the next step.

Using Your Local GOOS and GOARCH Environment Variables

Earlier, you ran the `go env GOOS GOARCH` command to find out what OS and architecture you were working on. When you ran the `go env` command, it looked for the two environment variables `GOOS` and `GOARCH`; if found, their values would be used, but if not found, then Go would set them with the information for the current platform. This means that you can change `GOOS` or `GOARCH` so that they do not default to your local OS and architecture.

The `go build` command behaves in a similar manner to the `go env` command. You can set either the `GOOS` or `GOARCH` environment variables to build for a different platform using `go build`.

If you are not using a Windows system, build a `windows` binary of **app** by setting the `GOOS` environment variable to `windows` when running the `go build` command:

```
GOOS=windows go build
```

Now list the files in your current directory:

```
ls
```

The output of listing the directory shows there is now an `app.exe` Windows executable in the project directory:

Output

```
app  app.exe  main.go  path.go  windows.go
```

Using the `file` command, you can get more information about this file, confirming its build:

```
file app.exe
```

You will receive:

Output

```
app.exe: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
```

You can also set one, or both environment variables at build time. Run the following:

```
GOOS=linux GOARCH=ppc64 go build
```

Your `app` executable will now be replaced by a file for a different architecture. Run the `file` command on this binary:

```
file app
```

You will receive output like the following:

```
app: ELF 64-bit MSB executable, 64-bit PowerPC or cisco 7500, version 1 (SYSV), statically linked, not stripped
```

By setting your local `GOOS` and `GOARCH` environment variables, you can now build binaries for any of Go's compatible platforms without a complicated configuration or setup. Next, you will use filename conventions to keep your files neatly organized and build for specific platforms automatically wihout build tags.

Using GOOS and GOARCH Filename Suffixes

As you saw earlier, the Go standard library makes heavy use of build tags to simplify code by separating out different platform implementations into different files. When you opened the `os/path_unix.go` file, there was a build tag that listed all of the possible combinations that are considered Unix-like platforms. The `os/path_windows.go` file, however, contained no build tags, because the suffix on the filename sufficed to tell Go which platform the file was meant for.

Let's look at the syntax of this feature. When naming a `.go` file, you can add `GOOS` and `GOARCH` as suffixes to the file's name in that order, separating the values by underscores (`_`). If you had a Go file named `filename.go`, you could specify the OS and architecture by

changing the filename to filename_**GOOS_GOARCH**.go. For example, if you wished to compile it for Windows with 64-bit ARM architecture, you would make the name of the file filename_windows_arm64.go. This naming convention helps keep code neatly organized.

Update your program to use the filename suffixes instead of build tags. First, rename the path.go and windows.go file to use the convention used in the os package:

```
mv path.go path_unix.go
mv windows.go path_windows.go
```

With the two filenames changed, you can remove the build tag you added to path_windows.go:

```
nano path_windows.go
```

Remove // +build windows so that your file looks like this:

path_windows.go

```
package main

const PathSeparator = "\\"
```

Save and exit from the file.

Because unix is not a valid GOOS, the _unix.go suffix has no meaning to the Go compiler. It does, however, convey the intended purpose of the file. Like the os/path_unix.go file, your path_unix.go file still needs to use build tags, so keep that file unchanged.

By using filename conventions, you removed uneeded build tags from your source code and made the filesystem cleaner and clearer.

Conclusion

The ability to generate binaries for multiple platforms that require no dependencies is a powerful feature of the Go toolchain. In this tutorial, you used this capability by adding build tags and filename suffixes to mark certain code snippets to only compile for certain architectures. You created your own platorm-dependent program, then manipulated the GOOS and GOARCH environment variables to generate binaries for platforms beyond your current platform. This is a valuable skill, because it is a common practice to have a continuous integration process that automatically runs through these environment variables to build binaries for all platforms.

For further study on go build, check out our Customizing Go Binaries with Build Tags tutorial. If you'd like to learn more about the Go programming language in general, check out the entire How To Code in Go series.

Using ldflags to Set Version Information for Go Applications

Written by Gopher Guides

When deploying applications into a production environment, building binaries with version information and other metadata will improve your monitoring, logging, and debugging processes by adding identifying information to help track your builds over time. This version information can often include highly dynamic data, such as build time, the machine or user building the binary, the Version Control System (VCS) commit ID it was built against, and more. Because these values are constantly changing, coding this data directly into the source code and modifying it before every new build is tedious and prone to error: Source files can move around and variables/constants may switch files throughout development, breaking the build process.

One way to solve this in Go is to use `-ldflags` with the `go build` command to insert dynamic information into the binary at build time, without the need for source code modification. In this flag, `ld` stands for linker, the program that links together the different pieces of the compiled source code into the final binary. `ldflags`, then, stands for linker flags. It is called this because it passes a flag to the

underlying Go toolchain linker, [`cmd/link`](), that allows you to change the values of imported packages at build time from the command line.

In this tutorial, you will use `-ldflags` to change the value of variables at build time and introduce your own dynamic information into a binary, using a sample application that prints version information to the screen.

Prerequisites

To follow the example in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment.

Building Your Sample Application

Before you can use `ldflags` to introduce dynamic data, you first need an application to insert the information into. In this step, you will make this application, which will at this stage only print static versioning information. Let's create that application now.

In your `src` directory, make a directory named after your application. This tutorial will use the application name **app**:

```
mkdir app
```

Change your working directory to this folder:

```
cd app
```

Next, using the text editor of your choice, create the entry point of your program, `main.go`:

```
nano main.go
```

Now, make your application print out version information by adding the following contents:

app/main.go

```go
package main

import (
    "fmt"
)

var Version = "development"

func main() {
    fmt.Println("Version:\t", Version)
}
```

Inside of the `main()` function, you declared the `Version` variable, then printed the string `Version:`, followed by a tab character, `\t`, and then the declared variable.

At this point, the variable `Version` is defined as `development`, which will be the default version for this app. Later on, you will change this value to be an official version number, arranged according to semantic versioning format.

Save and exit the file. Once this is done, build and run the application to confirm that it prints the correct version:

```
go build
./app
```

You will see the following output:

Output

```
Version:     development
```

You now have an application that prints default version information, but you do not yet have a way to pass in current version information at build time. In the next step, you will use `-ldflags` and `go build` to solve this problem.

Using `ldflags` with `go build`

As mentioned before, `ldflags` stands for linker flags, and is used to pass in flags to the underlying linker in the Go toolchain. This works according to the following syntax:

```
go build -ldflags="-flag"
```

In this example, we passed in `flag` to the underlying `go tool link` command that runs as a part of `go build`. This command uses double quotes around the contents passed to `ldflags` to avoid breaking characters in it, or characters that the command line might interpret as something other than what we want. From here, you could pass in many different `link` flags. For the purposes of this tutorial, we will use the -X flag to write information into the variable at

link time, followed by the package path to the variable and its new value:

```
go build -ldflags="-X
'package_path.variable_name=new_value'"
```

Inside the quotes, there is now the -X option and a key-value pair that represents the variable to be changed and its new value. The . character separates the package path and the variable name, and single quotes are used to avoid breaking characters in the key-value pair.

To replace the Version variable in your example application, use the syntax in the last command block to pass in a new value and build the new binary:

```
go build -ldflags="-X 'main.Version=v1.0.0'"
```

In this command, main is the package path of the Version variable, since this variable is in the main.go file. Version is the variable that you are writing to, and v1.0.0 is the new value.

In order to use ldflags, the value you want to change must exist and be a package level variable of type string. This variable can be either exported or unexported. The value cannot be a const or have its value set by the result of a function call. Fortunately, Version fits all of these requirements: It was already declared as a variable in the main.go file, and the current value (development) and the desired value (v1.0.0) are both strings.

Once your new **app** binary is built, run the application:

```
./app
```

You will receive the following output:

Output

```
Version:     v1.0.0
```

Using `-ldflags`, you have succesfully changed the `Version` variable from `development` to `v1.0.0`.

You have now modified a `string` variable inside of a simple application at build time. Using `ldflags`, you can embed version details, licensing information, and more into a binary ready for distribution, using only the command line.

In this example, the variable you changed was in the `main` program, reducing the difficulty of determining the path name. But sometimes the path to these variables is more complicated to find. In the next step, you will write values to variables in sub-packages to demonstrate the best way to determine more complex package paths.

Targeting Sub-Package Variables

In the last section, you manipulated the `Version` variable, which was at the top-level package of the application. But this is not always the case. Often it is more practical to place these variables in another package, since `main` is not an importable package. To simulate this in your example application, you will create a new sub-package, `app/build`,

that will store information about the time the binary was built and the name of the user that issued the build command.

To add a new sub-package, first add a new directory to your project named `build`:

```
mkdir -p build
```

Then create a new file named `build.go` to hold the new variables:

```
nano build/build.go
```

In your text editor, add new variables for `Time` and `User`:

app/build/build.go

```go
package build

var Time string

var User string
```

The `Time` variable will hold a string representation of the time when the binary was built. The `User` variable will hold the name of the user who built the binary. Since these two variables will always have values, you don't need to initialize these variables with default values like you did for `Version`.

Save and exit the file.

Next, open `main.go` to add these variables to your application:

```
nano main.go
```

Inside of `main.go`, add the following highlighted lines:

main.go

```
package main

import (
    "app/build"
    "fmt"
)

var Version = "development"

func main() {
    fmt.Println("Version:\t", Version)
    fmt.Println("build.Time:\t", build.Time)
    fmt.Println("build.User:\t", build.User)
}
```

In these lines, you first imported the `app/build` package, then printed `build.Time` and `build.User` in the same way you printed `Version`.

Save the file, then exit from your text editor.

Next, to target these variables with `ldflags`, you could use the import path `app/build` followed by `.User` or `.Time`,

since you already know the import path. However, to simulate a more complex situation in which the path to the variable is not evident, let's instead use the `nm` command in the Go tool chain.

The `go tool nm` command will output the symbols involved in a given executable, object file, or archive. In this case, a symbol refers to an object in the code, such as a defined or imported variable or function. By generating a symbol table with `nm` and using `grep` to search for a variable, you can quickly find information about its path.

Note: The `nm` command will not help you find the path of your variable if the package name has any non-ASCII characters, or a `"` or `%` character, as that is a limitation of the tool itself.

To use this command, first build the binary for **app**:

```
go build
```

Now that **app** is built, point the `nm` tool at it and search through the output:

```
go tool nm ./app | grep app
```

When run, the `nm` tool will output a lot of data. Because of this, the preceding command used `|` to pipe the output to the `grep` command, which then searched for terms that had the top-level **app** in the title.

You will receive output similar to this:

Output

```
  55d2c0 D app/build.Time
  55d2d0 D app/build.User
  4069a0 T runtime.appendIntStr
  462580 T strconv.appendEscapedRune
. . .
```

In this case, the first two lines of the result set contain the paths to the two variables you are looking for: `app/build.Time` and `app/build.User`.

Now that you know the paths, build the application again, this time changing `Version`, `User`, and `Time` at build time. To do this, pass multiple `-X` flags to `-ldflags`:

```
go build -v -ldflags="-X 'main.Version=v1.0.0' -X
'app/build.User=$(id -u -n)' -X
'app/build.Time=$(date)'"
```

Here you passed in the `id -u -n` Bash command to list the current user, and the `date` command to list the current date.

Once the executable is built, run the program:

```
./app
```

This command, when run on a Unix system, will generate similar output to the following:

Output

```
Version:    v1.0.0
build.Time: Fri Oct  4 19:49:19 UTC 2019
build.User: sammy
```

Now you have a binary that contains versioning and build information that can provide vital assistance in production when resolving issues.

Conclusion

This tutorial showed how, when applied correctly, `ldflags` can be a powerful tool for injecting valuable information into binaries at build time. This way, you can control feature flags, environment information, versioning information, and more without introducing changes to your source code. By adding `ldflags` to your current build workflow you can maximize the benefits of Go's self-contained binary distribution format.

If you would like to learn more about the Go programming language, check out our full How To Code in Go series. If you are looking for more solutions for version control, try our How To Use Git reference guide.

How To Use the Flag Package in Go

Written by Gopher Guides

Command-line utilities are rarely useful out of the box without additional configuration. Good defaults are important, but useful utilities need to accept configuration from users. On most platforms, command-line utilities accept flags to customize the command's execution. Flags are key-value delimited strings added after the name of the command. Go lets you craft command-line utilities that accept flags by using the `flag` package from the standard library.

In this tutorial you'll explore various ways to use the `flag` package to build different kinds of command-line utilities. You'll use a flag to control program output, introduce positional arguments where you mix flags and other data, and then implement sub-commands.

Using a Flag to Change a Program's Behavior

Using the `flag` package involves three steps: First, define variables to capture flag values, then define the flags your Go application will use, and finally, parse the flags provided to the application upon execution. Most of the functions within the `flag` package are concerned with defining flags and binding them to variables that you have defined. The parsing phase is handled by the `Parse()` function.

To illustrate, you'll create a program that defines a Boolean flag that changes the message that will be printed to standard output. If there's a `-color` flag provided, the program will print a message in blue. If no flag is provided, the message will be printed without any color.

Create a new file called `boolean.go`:

```
nano boolean.go
```

Add the following code to the file to create the program:

boolean.go

```go
package main

import (
	"flag"
	"fmt"
)

type Color string

const (
	ColorBlack  Color = "\u001b[30m"
	ColorRed          = "\u001b[31m"
	ColorGreen        = "\u001b[32m"
	ColorYellow       = "\u001b[33m"
	ColorBlue         = "\u001b[34m"
	ColorReset        = "\u001b[0m"
)

func colorize(color Color, message string) {
	fmt.Println(string(color), message, string(ColorReset))
}

func main() {
	useColor := flag.Bool("color", false, "display colorized output")
	flag.Parse()

	if *useColor {
		colorize(ColorBlue, "Hello, DigitalOcean!")
		return
	}
	fmt.Println("Hello, DigitalOcean!")
}
```

This example uses ANSI Escape Sequences to instruct the terminal to display colorized output. These are specialized sequences of characters, so it makes sense to define a new type for them. In this example, we've called that type `Color`, and defined the type as a `string`. We then define a palette of colors to use in the `const` block that follows. The `colorize` function defined after the `const` block accepts one of these `Color` constants and a `string` variable for the message to colorize. It then instructs the terminal to change

color by first printing the escape sequence for the color requested, then prints the message, and finally requests that the terminal reset its color by printing the special color reset sequence.

Within `main`, we use the `flag.Bool` function to define a Boolean flag called `color`. The second parameter to this function, `false`, sets the default value for this flag when it is not provided. Contrary to expectations you may have, setting this to `true` does not invert the behavior such that providing a flag will cause it to become false. Consequently, the value of this parameter is almost always `false` with Boolean flags.

The final parameter is a string of documentation that can be printed as a usage message. The value returned from this function is a pointer to a `bool`. The `flag.Parse` function on the next line uses this pointer to set the `bool` variable based on the flags passed in by the user. We are then able to check the value of this `bool` pointer by dereferencing the pointer. More information about pointer variables can be found in the tutorial on pointers. Using this Boolean value, we can then call `colorize` when the `-color` flag is set, and call the `fmt.Println` variable when the flag is absent.

Save the file and run the program without any flags:

```
go run boolean.go
```

You'll see the following output:

Output

```
Hello, DigitalOcean!
```

Now run this program again with the `-color` flag:

```
go run boolean.go -color
```

The output will be the same text, but this time in the color blue.

Flags are not the only values passed to commands. You might also send file names or other data.

Working with Positional Arguments

Typically commands will take a number of arguments that act as the subject of the command's focus. For example, the `head` command, which prints the first lines of a file, is often invoked as `head example.txt`. The file `example.txt` is a positional argument in the invocation of the `head` command.

The `Parse()` function will continue to parse flags that it encounters until it detects a non-flag argument. The `flag` package makes these available through the `Args()` and `Arg()` functions.

To illustrate this, you'll build a simplified re-implementation of the `head` command, which displays the first several lines of a given file:

Create a new file called `head.go` and add the following code:

head.go

```go
package main

import (
	"bufio"
	"flag"
	"fmt"
	"io"
	"os"
)

func main() {
	var count int
	flag.IntVar(&count, "n", 5, "number of lines to read from the file")
	flag.Parse()

	var in io.Reader
	if filename := flag.Arg(0); filename != "" {
		f, err := os.Open(filename)
		if err != nil {
			fmt.Println("error opening file: err:", err)
			os.Exit(1)
		}
		defer f.Close()

		in = f
	} else {
		in = os.Stdin
	}

	buf := bufio.NewScanner(in)

	for i := 0; i < count; i++ {
		if !buf.Scan() {
			break
		}
		fmt.Println(buf.Text())
	}

	if err := buf.Err(); err != nil {
```

```
        fmt.Fprintln(os.Stderr, "error reading: err:", err)
    }
}
```

First, we define a `count` variable to hold the number of lines the program should read from the file. We then define the `-n` flag using `flag.IntVar`, mirroring the behavior of the original `head` program. This function allows us to pass our own pointer to a variable in contrast to the `flag` functions that do not have the `Var` suffix. Apart from this difference, the rest of the parameters to `flag.IntVar` follow its `flag.Int` counterpart: the flag name, a default value, and a description. As in the previous example, we then call `flag.Parse()` to process the user's input.

The next section reads the file. We first define an `io.Reader` variable that will either be set to the file requested by the user, or standard input passed to the program. Within the `if` statement, we use the `flag.Arg` function to access the first positional argument after all flags. If the user supplied a file name, this will be set. Otherwise, it will be the empty string (`""`). When a filename is present, we use the `os.Open` function to open that file and set the `io.Reader` we defined before to that file. Otherwise, we use `os.Stdin` to read from standard input.

The final section uses a `*bufio.Scanner` created with `bufio.NewScanner` to read lines from the `io.Reader` variable `in`. We iterate up to the value of `count` using a for loop, calling `break` if scanning the line with `buf.Scan` produces a `false` value, indicating that the number of lines is less than the number requested by the user.

Run this program and display the contents of the file you just wrote by using `head.go` as the file argument:

```
go run head.go -- head.go
```

The `--` separator is a special flag recognized by the `flag` package which indicates that no more flag arguments follow. When you run this command, you receive the following output:

Output

```
package main

import (
        "bufio"
        "flag"
```

Use the `-n` flag you defined to adjust the amount of output:

```
go run head.go -n 1 head.go
```

This outputs only the package statement:

Output

```
package main
```

Finally, when the program detects that no positional arguments were supplied, it reads input from standard input, just like `head`. Try running this command:

```
echo "fish\nlobsters\nsharks\nminnows" | go run head.go -n 3
```

You'll see the output:

Output

```
fish
lobsters
sharks
```

The behavior of the `flag` functions you've seen so far has been limited to examining the entire command invocation. You don't always want this behavior, especially if you're writing a command line tool that supports sub-commands.

Using FlagSet to Implement Sub-commands

Modern command-line applications often implement "sub-commands" to bundle a suite of tools under a single command. The most well-known tool that uses this pattern is `git`. When examining a command like `git init`, `git` is the command and `init` is the sub-command of `git`. One notable feature of sub-commands is that each sub-command can have its own collection of flags.

Go applications can support sub-commands with their own set of flags using the `flag.(*FlagSet)` type. To illustrate this, create a program that implements a command using two sub-commands with different flags.

Create a new file called `subcommand.go` and add the following content to the file:

```go
package main

import (
    "errors"
    "flag"
    "fmt"
    "os"
)

func NewGreetCommand() *GreetCommand {
    gc := &GreetCommand{
        fs: flag.NewFlagSet("greet", flag.ContinueOnError),
    }
```

```go
    gc.fs.StringVar(&gc.name, "name", "World", "name of the person to be gree

    return gc
}

type GreetCommand struct {
    fs *flag.FlagSet

    name string
}

func (g *GreetCommand) Name() string {
    return g.fs.Name()
}

func (g *GreetCommand) Init(args []string) error {
    return g.fs.Parse(args)
}

func (g *GreetCommand) Run() error {
    fmt.Println("Hello", g.name, "!")
    return nil
}

type Runner interface {
    Init([]string) error
    Run() error
    Name() string
}

func root(args []string) error {
    if len(args) < 1 {
        return errors.New("You must pass a sub-command")
    }

    cmds := []Runner{
        NewGreetCommand(),
    }
```

```
    subcommand := os.Args[1]

    for _, cmd := range cmds {
        if cmd.Name() == subcommand {
            cmd.Init(os.Args[2:])
            return cmd.Run()
        }
    }

    return fmt.Errorf("Unknown subcommand: %s", subcommand)
}

func main() {
    if err := root(os.Args[1:]); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

This program is divided into a few parts: the `main` function, the `root` function, and the individual functions to implement the sub-command. The `main` function handles errors returned from commands. If any function returns an error, the `if` statement will catch it, print the error, and the program will exit with a status code of `1`, indicating that an error occurred to the rest of the operating system. Within `main`, we pass all of the arguments the program was invoked with to `root`. We remove the first argument, which is the name of the program (in the previous examples `./subcommand`) by slicing `os.Args` first.

The `root` function defines `[]Runner`, where all sub-commands would be defined. `Runner` is an interface for sub-commands that allows `root` to retrieve the name of the sub-command using `Name()` and compare it against the contents `subcommand` variable. Once the correct sub-command is located after iterating through the `cmds` variable we initialize the sub-command with the rest of the arguments and invoke that command's `Run()` method.

We only define one sub-command, though this framework would easily allow us to create others. The `GreetCommand` is instantiated using `NewGreetCommand` where we create a new `*flag.FlagSet` using `flag.NewFlagSet`. `flag.NewFlagSet` takes two arguments: a name for the flag set, and a strategy for reporting parsing errors. The `*flag.FlagSet`'s name is accessible using the `flag.(*FlagSet).Name` method. We use this in the `(*GreetCommand).Name()` method so the name of the sub-command matches the name we gave to the `*flag.FlagSet`. `NewGreetCommand` also defines a `-name` flag in a similar way to previous examples, but it instead calls this as a method off the `*flag.FlagSet` field of the

*GreetCommand, gc.fs. When root calls the Init() method of the *GreetCommand, we pass the arguments provided to the Parse method of the *flag.FlagSet field.

It will be easier to see sub-commands if you build this program and then run it. Build the program:

```
go build subcommand.go
```

Now run the program with no arguments:

```
./subcommand
```

You'll see this output:

Output

```
You must pass a sub-command
```

Now run the program with the greet sub-command:

```
./subcommand greet
```

This produces the following output:

Output

```
Hello World !
```

`

Now use the -name flag with greet to specify a name:

```
./subcommand greet -name Sammy
```

You'll see this output from the program:

Output

```
Hello Sammy !
```

This example illustrates some principles behind how larger command line applications could be structured in Go. FlagSets are designed to give developers more control over where and how flags are processed by the flag parsing logic.

Conclusion

Flags make your applications more useful in more contexts because they give your users control over how the programs execute. It's important to give users useful defaults, but you should give them the opportunity to override settings that don't work for their situation. You've seen that the flag package offers flexible choices to present configuration options to your users. You can choose a few simple flags, or build an extensible suite of sub-commands. In either case, using the flag package will help you build utilities in the style of the long history of flexible and scriptable command line tools.

To learn more about the Go programming language, check out our full How To Code in Go series.